Assignment # 2-B

PART A - ENCRYPTION

Get ready for coded secret message!

BACKGROUND

Supposedly, we wanted to 'encrypt' (i.e., conceal in a reversible way) confidential messages by shifting each letter therein by some number of places. For instance, he might write A as B, B as C, C as D, …, and, wrapping around alphabetically, Z as A. And so, to say HELLO to someone, we might write IFMMP instead. Upon receiving such messages from us, recipients would have to "decrypt" them by shifting letters in the opposite direction by the same number of places.

The secrecy of this 'cryptography system' rely only on us and the recipients knowing the secret, that is, the number of places by which we had shifted the letters (e.g. +1 or +13). Well, not particularly secure by modern standards, but, hey, if you're perhaps the first in the world to do it, pretty sure it's secure!

'Un-encrypted' text is generally called **'plaintext'**. Encrypted text is generally called **'ciphertext'**. And the secret used is called a **key**.

To be clear, here's how encrypting HELLO with a key of 1 yields IFMMP :

Plaintext	H	E	L	L	O
Key: +	1	1	1	1	1
Ciphertext	I	F	M	M	P

More formally, Caesar's algorithm or cipher (Yes, Caesar himself used this system in Roman days) encrypts messages by 'rotating' or converting each letter by k positions.

More formally, if p is some plaintext (i.e., an unencrypted message), p_i is the i^{th} character in p, and k is a secret key (i.e., a non-negative integer), then each letter, c_i, in the ciphertext, c, is computed as

$$c_i = (p_i + k) \% 26 \qquad \textbf{(FORMULA 1)}$$

or in coding formula : c[i] = (p[i]+k) % 26 where i is index number wherein % 26 here means 'the remainder obtained when dividing by 26.' This formula perhaps makes the cipher seem more complicated than it is, but it's really just a concise way of expressing the algorithm precisely.

Indeed, for the sake of discussion, think of 'A' (or 'a") as 0, 'B' (or 'b') as 1, …, 'H' (or 'h') as 7, 'I' (or 'i') as 8, …, and 'Z' (or 'z') as 25. See **TABLE 1** below:

char	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
char	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
value	0	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

Suppose that Caesar just wants to say "Hi" to someone confidentially using, this time, a key, k = 3. And so his plaintext, p variable, is "Hi" , in which his plaintext's first character, p[0], is H (aka 7 ~ see Table 1 above), and his plaintext's second character, p[1], is i (aka 8 ~ see Table 2 above). Thus, with key -> +3, then his ciphertext variable of c, has its first character, which is c[0], becomes K (aka 7 + 3 = 10 ~ see Table 1 above), and subsequently his ciphertext's second character, c[1], is thus l (aka 8 + 3 = 11 ~ see Table 1 above). Make sense?

In conclusion, for us to use FORMULA 1 above when converting the characters of our string or plaintext, according to the key given, we need to somehow ensure that we have an alphabetical numbering system where the value of each character – regardless uppercase and lowercase, follows the value set in Table 1. How do we that in coding? Hint # 1: Well, remember the ASCII table where every characters have their own integer decimal based on base 10 system? Perhaps, we should use some simple mathematical algorithm to align the ASCII integer number with the number define in table 1 both for uppercase and lowercase alphabets. This will be explained in detail later.

EXPECTATIONS ON CODE OUTPUTS

Let us now write a program called encrypt.c that enables you to encrypt messages using Caesar's ciphering algorithm. At the time when user executes the program, they should decide, by providing a command-line argument, what the key should be in the secret message they'll provide at runtime using the get_string function.

Here are few examples of how the program might work. For example, if the user inputs a command-line argument during execution of encrypt.c, a key of 1 and a plaintext of HELLO obtained from the runtime of get_string function implemented in the code of encrypt.c program, the output will be such as shown:

```
$./encrypt 1
Plaintext:  HELLO
Ciphertext: IFMMP
```

Note that for window's user:

```
$./encrypt.exe 1
Plaintext:  HELLO
Ciphertext: IFMMP
```

Here's how the program might work if users provide key of 13 and plaintext of "Hello, World!":

```
$./encrypt 13
Plaintext:  Hello, World!
Ciphertext: Uryyb, Jbeyq!
```

Again, for window's user:

```
$./encrypt.exe 13
Plaintext:  Hello, World!
Ciphertext: Uryyb, Jbeyq!
```

Here's another last example if users still provide key of 13 at command-argument with plaintext obtained from get_string function at runtime of "Be sure to eat your Breakfast"

```
$./encrypt 13
Plaintext:  1. Be sure to eat your Breakfast
Ciphertext: 1. Or fher gb rng lbhe Oernxsnfg
```

There are several things we need to observe based on the 3 output examples provided here and those will be explained on the specification requirements of our encrypt.c code.

CODE'S OUTPUT SPECIFICATIONS

1. The key is a positive or non-negative number. The value can range up to maximum allowable value for integer.

2. Uppercase letters remain uppercase and lowercase remain lowercase. Thus, the case must be preserved. How to achieved this? Hint # 2: First, of course, just like what has been shown in our lecture, every individual characters of our string or plaintext will be evaluated using its index number and this is achieved using the for loop statement. The number of iterations set in our for loop statement depends on the length of our string or plaintext, which can be found by using the strlen function offered by string.h library. Hint # 3: Then, for every characters of our string or plaintext evaluated, we may want to use isupper and islower function provided by ctype.h library as a condition in our if statement to check wether the character is uppercase or not. In either case, the conversion or 'rotation' of the character will still take place. That means the conversion procedure for uppercase alphabets is applicable to lowercase alphabets as well.

3. At the same time, notice that non-alphabetical characters such as whitespace in between words, exclamation mark, period, comma and even numbers will not be converted. How do we achieved this? Hint # 4: Again, ctype.h library provides this one nice function that we can use to check whether the characters of our plaintext are alphabetical character or not and such function is called isalpha. If the condition is true, then necessary conversion step to our alphabetical characters will be proceeded and if not, meaning false condition, we straight away print that non-alphabetical characters to the console screen.

4. Notice that there are characters, regardless uppercase or lowercase, when added or converted by the key's number exceed 26 (or 25 as per TABLE 1), to which the conversion is wrapped around back to the starting point from value 0 for the uppercase 'A' and lowercase 'a'. What does this mean? Let's take a look on the second example of code's output where the plaintext is "Hello,

World! And let us focus on the first character of the second word, which is 'W'. After it has been checked that this 'W' is indeed an alphabetical character and also an uppercase letter, this character 'W' will then be converted or rotated to another character shifted by 13 to the right or in another word, the value of character 'W' is added by 13, which as per Table 1, the original value of 'W' is 22 and when it is added to 13, it should 35 right? But the max number in Table 1 is only 25 representing the 26th characters or the uppercase 'Z' character. Thus, from 22 to 25, we have shifted by 3 and the balance from 13 is 10. Reaching 'Z' when shifted by 3, it has to go back to the beginning of Table 1 and proceed by shifting 10 places to the right, which will bring us to alphabet 'J'. BUT, how do achieve this wrapped around process? Again, this goes back to our Hint # 1 where in our code, before the conversion process of the individual character, we need to establish a numbering system for uppercase letters as per Table 1 and also the same numbering system for the lowercase letter as per Table 1 too. How do we establish such numbering system? Hint # 5: As for uppercase alphabets and as per ASCII table, we simply deduct every characters from 'A', 'B', .. to... 'Z' with alphabet 'A' and thus, we will get the numbering system as per Table 1 for uppercase letters. For example, as per ASCII Table, uppercase letter 'A' has its decimal value of 65 and thus, by deducting itself, i.e. 65 - 65, we get 0 value and for 'B', which has a decimal value of 66 and thus, deduct 'A' from itself, i.e 66 - 65, we get value of 1 for uppercase letter 'B' as per our Table 1. Therefore, whenever we get uppercase alphabetical characters obtained from our plaintext, it needs to be deducted first with uppercase letter 'A' before the conversion. Once deducted, Hint # 6: we can now use FORMULA 1 above to properly convert the character's value according to the key number. Based on that formula, k represents the positive key's number while our individual evaluated character's (from plaintext) value is represented by p_i where again, i stands for the index number representing the position of that character in our plaintext. By using that formula, we will get our converted or rotated ciphered character, c_i. The formula itself, will ensure any character that goes over the value of 25 as per Table 1 will automatically be wrapped around back to the start of our Table 1 and automatically shifted to actual value of converted character. Once we get the converted or rotated value from the formula, it is important that this converted or rotated value be transformed back to the actual value defined by the ASCII Table. Otherwise, we will not be able to print correctly the alphabetical uppercase letter for the converted/rotated character. Thus, Hint #7: every converted value obtained by formula 1 must be added back with uppercase character 'A'. On final note, logically the same process should be applied for lowercase alphabets, right? However, just note that lowercase alphabet 'a' has different ASCII value than uppercase 'A'.

Now, let's have a look at the specifications/requirements of the code itself.

CODE'S SPECIFICATIONS

1. The C program's name for encryption shall named as 'encrypt.c'. The output machine code shall simply be named as 'encrypt' or for window's user, as 'encrypt.exe'.

2. The plaintext shall be obtained using the get_string() function provided by the cs50 library during runtime. Nevertheless, the prompt message will need to be as follow: get_string("Plaintext: ") where we need to allocate two whitespace after colon.

3. In printing out the converted ciphertext, we need to print out a word 'Ciphertext:' using the printf function, which for example, printf("Ciphertext: ") where this time, only one whitespace is provided after the colon. Then, the individual converted alphabetical characters and non-rotated non-alphabetical characters are printed out one by one according to the iteration index number i. See our lecture's video how this is achieved.

4. The key's number must be obtained from the command-line argument during execution of the output machine code. See our video lecture's how this can be achieved in your code where in the parenthesis of our int main program, Hint # 8: we have, for example, int main(int argc, string argv[]).

5. There are several requirements applicable to the input provided through command-line argument. First, the code must be able to check that the command-line shall consist only 2 arguments. Anything less, for example, just 1 argument or more than 2 arguments when executing the code, the code must detect it as an error. To this, some error message will be printed out using printf function and furthermore, return 1 statement will be used to exit from the main function, thus terminating the program. Watch our video lecture to see how these all are implemented. However, how specifically do we implement the check? Hint # 9: Use a condition that employs integer argc variable where it can be used to check the number of arguments provided by users in the command-line argument. Now, what will the error message suppose to be if the condition is not met? In this case, it should print out a short message like "Usage: ./encrypt key", or for window's user: "Usage .\ encrypt.exe key". Here are several scenario related to the incorrect and correct command-line argument:

   Example 1: only 1 argument – incorrect input

   ```
   $./encrypt
   Usage: ./encrypt key
   ```

   Example 2: more than 2 arguments – incorrect input

   ```
   $./encrypt 20 5
   Usage: ./encrypt key
   ```

   Example 3: 2 arguments, which is a correct input and thus, subsequently prompting the user to enter plaintext input,

   ```
   $./encrypt 7
   Plaintext:
   ```

6. Next, in the case of where the user entered correct number of arguments in the command-line argument, it is safe to assume that there will be a time when the user may not enter a number as the key or perhaps a negative number as a key. Thus, we must provide some check against such conditions as well. Thus, what kind of check in your code must be provided? Hint #10: Assuming we have entered correct number of argument, the next step is to test wether that second argument or the argv[1] is a digit character or not. Remember that the input from command-line argument is a string type value and not a

numerical value when we declare the argv[] array in the parenthesis of int main() function as a string type array variable. Because of this, we need to check wether the string or argv[1] has digit characters and this is achieved by a function called isdigit(), which is provided by the ctype.h library. It is important to note that isdigit only check individual character and not the whole string. Thus, a loop is needed. Next, if the condition of isdigit is false, i.e. either all or some of the characters of argv[1] is not a digit, then an error message will be printed out and return 1 statement is enforced in the code again where this will terminate the program by exiting the main function. The error message should be as follow: "Usage: ./encrypt key (key is digit)". Specific examples:

```
$./encrypt 2A1
Usage: ./encrypt key (key is positive digit)

$./encrypt banana
Usage: ./encrypt key (key is positive digit)

$./encrypt -4
Usage: ./encrypt key (key is positive digit)
```

7. If key is indeed alphanumeric or digit character, then we should proceed to the next step of the code where we need to convert the key's digit from string type value to actual integer type numerical value so that it can be used in FORMULA 1. Hint# 11: To convert digit character to integer number, we can use atoi function provided by the stdlib.h library.

FINAL ADVISES

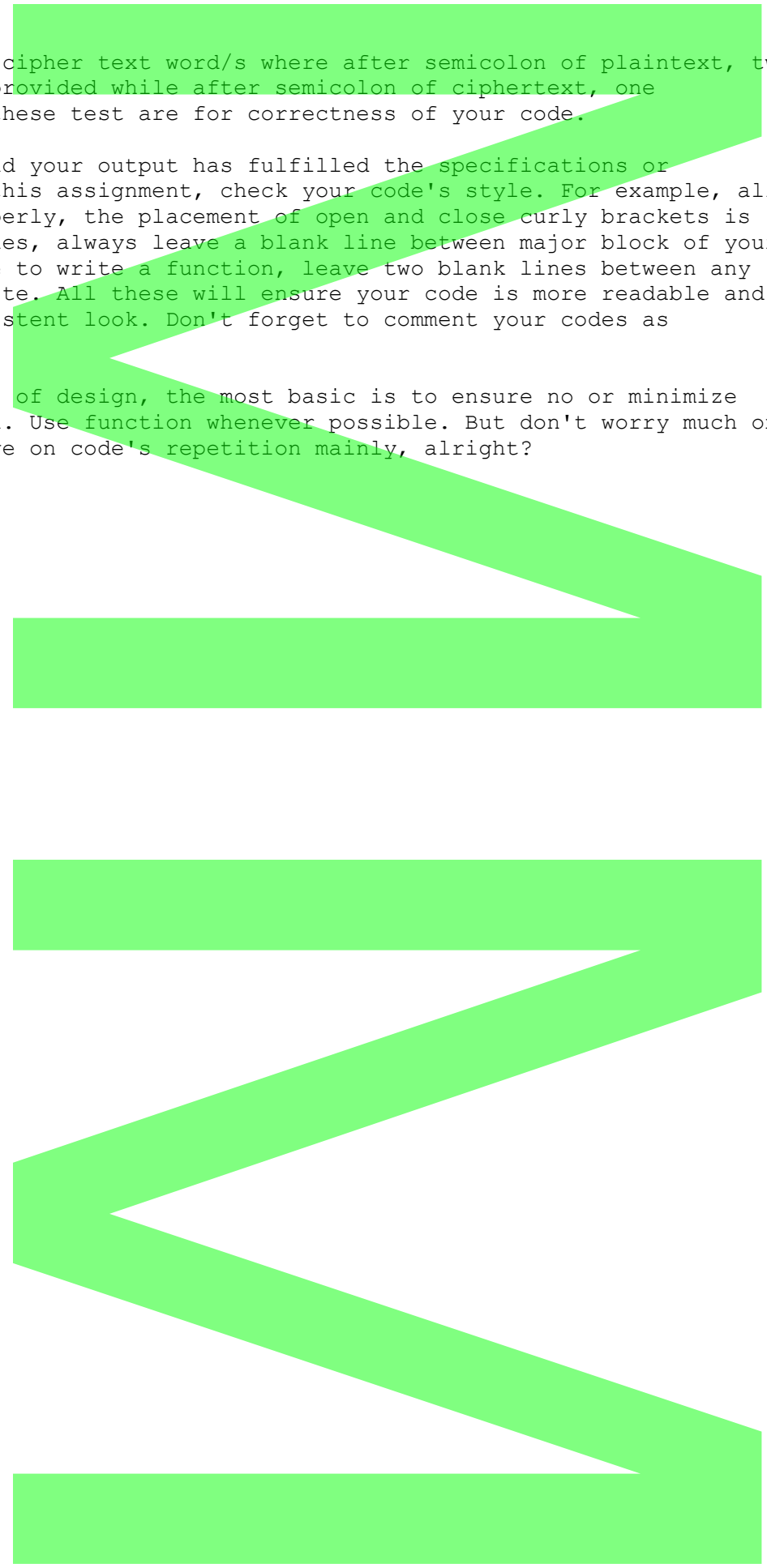
1. So how do we write our main program in encrypt.c? Well, it is advisable to write down the pseudocode first based on your understandings from this reading as well as from your discussion with the lecturer. Remember, pseudocode is just like step-by-step instructions to achieve the desired output of the code.

2. Always test your code thoroughly to see if it works according the specifications or requirements of this assignment. First, test your code on the check of getting the right number of argument. During execution of the output code, test without command-line argument input and then, test it with more than 2 arguments. Our expectation is to get the error message, program terminated with echo value of 1. Second, test your code with the correct number of argument, which is 2 argument but the second argument, which is the key needed for the encryption, is not a digit. Test it with alphabetical character or a combination of alphanumeric and alphabetical characters. Again, our expectation is to get the desired error message and program terminated with echo value of 1. Then, the third test is we enter the correct number of argument and correct digit character so that we can now test whether we get the right output for the conversion or not, i.e. we must ensure that all uppercase and lowercase of the characters of plaintext are preserved after conversion. Then we must ensure, all non-alphabetical characters are not converted and remain as it is and last in this third test, the converted text or every individual characters are converted as per according to the FORMULA 1 of Caesar's algorithm and the key number given. Once all is correct, do the check on the appearance of your output code, that is, as per example shown here and the alignment requirement of starting

of plaintext and cipher text word/s where after semicolon of plaintext, two whitespaces are provided while after semicolon of ciphertext, one whitespace. All these test are for correctness of your code.

3. Once your code and your output has fulfilled the specifications or requirements of this assignment, check your code's style. For example, all are indented properly, the placement of open and close curly brackets is always on new lines, always leave a blank line between major block of your code. If you were to write a function, leave two blank lines between any functions you write. All these will ensure your code is more readable and has cleaner consistent look. Don't forget to comment your codes as necessary.

4. Finally, in terms of design, the most basic is to ensure no or minimize code's repetition. Use function whenever possible. But don't worry much on this, just observe on code's repetition mainly, alright?



PART B - 8BITS

Get ready for another coded secret message!

BACKGROUNDS

This following assignment requires us to think about number base systems. For a start, the simplest base numbering system is base-1, or unary numbering system. For example, to write a number, N, in base-1, we would simply write N consecutive 1 s. So, the number 4 in base-1 would be written as 1111, and the number 12 as 111111111111 . Think of it like counting on your fingers or tallying up a score with marks on a board.

Thus, we might see why base-1 isn't used much nowadays. The numbers get rather long for larger number! Instead, a common convention is to use a base-10, or decimal number. In base-10, each digit is multiplied by some power of 10 when representing some larger numbers. For instance, 123 is short for as the following:

Bases of 10		10^2		10^1		10^0
Base-10 Numbers (0-9)	x	1	x	2	x	3
Equivalent decimal number		100	+	20	+	3 = **123**

More importantly, changing the base number, is as simple as by just changing the original base number to a different base number. For instance, if you wrote 123 in base-4, the equivalent decimal number, i.e. base-10 number, we would really be writing, $123 = 1 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0$, which is equal to the decimal number 27. For much clear example:

Bases of 4		4^2		4^1		4^0
Base-4 Numbers (0-4)	x	1	x	2	x	3
Equivalent decimal number		16	+	8	+	3 = **27**

Computers, though, use base-2, or binary. In binary, writing 123 would be a mistake, since binary numbers can only have 0 s and 1 s. But the process of figuring out exactly what decimal number a binary number stands for is exactly the same. For instance, the number 10101 in base-2 represents $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, which is equal to the decimal number 21. Again for clearer example:

Bases of 2		2^4		2^3		2^2		2^1		2^0
Base-2 Numbers	x	1	x	0	x	1	x	0	x	1
Equivalent decimal number		16	+	0	+	4	+	0	+	1 = **21**

Now that you know about base numbering system, let's dive in into the assignment requirements.

SO, WHAT'S THE INPUT AND OUTPUT OF OUR CODES?

Well, we are going to encode a message where for each individual character in the message, they will be converted into a series of 8 light bulbs representing the 8-bits binary number of each character. Wait, what? Well, every alphabetical characters has its own equivalent decimal number represented in the ASCII table. Thus, the code will convert this decimal number into 8-bits binary number but instead of binary number, a series of 8 bulbs is used for every character in our message. Thus basically, our code will get user input message and then, read its individual characters and turn it into encoded message in the form of 8 bulbs in series of on and off for 1 and 0 respectively.

Now, let's write a program called '8bits.c' that takes a message and converts it to a set of 8 bulbs that we could show it to the console screen. This, we will do it in three steps:

1. The first step consists of turning the alphabets of text into decimal numbers. Let's say we want to encode the message HI!. Using ASCII table (https://simple.m.wikipedia.org/wiki/File:ASCII-Table.svg), we find that H is represented by the decimal number 72 , I is represented by 73 , and ! is represented by 33.

2. The next step involves taking our decimal numbers (like 72 , 73 , and 33) and converting them into equivalent binary numbers, which only use 0s and 1s. For the sake of having a consistent number of bits in each of our binary numbers, assume that each decimal is represented with 8 bits. 72 is 01001000 , 73 is 01001001 , and 33 is 00100001.

3. Lastly, we'll interpret these binary numbers as instructions for the light bulbs on console screen where 0 is off and 1 is on. (You'll find that 8bits.c includes a print_bulb function that has been partially implemented for you, which takes in a 0 or 1 character and outputs emoji representing light bulbs.) We said partially completed right? Well, that means you get to work with different emojis to represent the on and off of every bit.

Here's an example of how the completed program might work where we will print one byte (8 bits) per line for each character of the message for clarity. In this example, emoji represents 'on' or '1' while emoji represents 'off' or '0':

```
$./8bits
Message: HI!
```

We can actually check our encoded message wether it is correct or not, right!? Just convert the as '1' and emoji as '0', which we have the following:

```
01001000
01001001
00100001
```

and as per ASCII table, they are indeed the correct representation of the individual character of H, I and !.

One thing to note that every character is encoded accordingly including non-alphabetical character such as whitespace, which as ASCII decimal number of 32 and binary as '00100000'. Cool eh!

CODE SPECIFICATIONS

Design and implement a program that converts text into instructions for the strip of bulbs on the console screen as follows:

1. Implement the program in a file called '8bits.c'.

2. The program will prompt user for a message or string using get_string function. The prompt instruction shall be as follow: "Message: " (one whitespace after colon).

3. The program must then convert the string into a series of 8-bit binary characters, one for each message's character in the string. Note that, it is an 8-bit binary characters, not an 8-bit binary numbers. The reason to this is that we may not be able get a full 8-bit binary number when converting the ASCII decimal number of the individual message's character to its binary form, to which we need to prepend (not append) the '0' character to the encoded binary form if not complete. And this can only be done if the binary representation is a string consists of 9 characters and not numbers. Wait, what? 9? Yup! 9 since string must have '\0' as the last character and thus, 8 characters of either 1s and 0s with '\0' gives us a total of 9 characters

4. The program then send the 8-bit binary characters to print_bulb function as to print a series of zeros and ones in terms of 2 coloured emojis, which represent off and on light bulbs respectively. The two colored-emoji will depend on your unique number assigned to you based on your full name including father's name.

5. Each 'byte' of 8 symbols should be printed on its own line when outputted and thus, there should be a '\n' after each last 'byte' of 8 symbols has been printed out as well.

HINTS

1. ITERATION OVER INDIVIDUAL CHARACTER IN STRING

Since we want to encode every character of the message (string) with a series of 8-bits binary form that can be read by print_bulb function, we then need to iterate over every character in our string's message. Hint #1: To do this, we need to use strlen function offered by string.h library.

Once we get the length of our string, we can use that information as a condition in our for loop, setting the limit, the number of times allowable to loop as to iterate every individual character using index number (See our video lecture if not clear).

Thus, we can now do the conversion for every individual character in our message.

2. CONVERTING ASCII DECIMAL NUMBER OF CHARACTER TO 8-BIT BINARY CHARACTERS

Now, let's understand first the mathematical procedures on how to convert decimal number to binary number. Let us walk through an example with the number 4. How would you convert 4 to binary?

Hint # 2: Start by considering the *right-most bit*. Hint # 3: First, divide 4 by 2 since 2 is the bases of binary number:

4 / 2 = 2 with remainder = 0

From above operation, two things we should be observed. Firstly, the result of that division operation is 2. Secondly, that division operation has no remainder or the remainder is 0. Thus, based on that remainder value, the first right-most bit is equal to 0.

Next, we need to perform another division by 2 operation. Hint # 4: This further division is needed because we have not reached 0 value resulted from division operation. Thus,

2 / 2 = 1 with remainder = 0

Again, we have no remainder from this operation or the remainder is equal to 0, which means that the *second right-most bit* is also 0. So by now we have a binary number of '00' at our hands.

Since, we have 1 as the result from the last division, which is not 0 value, we need to proceed to the next division by 2 again:

1 / 2 = 0 with remainder = 1

We get remainder 1 because 1 cannot be divided by 2 from integer operation stand point. Thus, the *third right-most bit* is 1. Now, do we have to further divide by 2 from here? Nope, because we have reached to 0 value from the last division operation between 2 integers. Of course, one will say that 1 divided by is 0.5 but again, Hint #5: this is an integer division operation where 0.5 is truncated to 0. Thus, we stop here and our binary number shall now be '100'.

However, please be noted that we have yet to get the complete 8-bits binary characters based on this example. That means, we need to have another structure that can actually preprend five 0s to that binary number. This will be explained later.

Now, let's take a look at how all of the above can be implemented in C programming language.

a) First, we need to understand that our 8-bit binary form is actually a string variable consisting of 9 characters or 9 elements, where the first 8 characters, could either be '1' or '0' and one more character at the end of our string, shall be the nul character - '\0'. Let's consider the actual 8-bits binary characters for 'H', which may look like the following:

string binary[9]								
0	1	0	0	1	0	0	0	\0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

The first row of the table shown suggested that a string type variable called binary having size of 9 element (it could have different name depending on user's choice) is used to store the 8-bits binary character; the second row is the binary characters consisting either '1' or '0'; and the third row is the index number of that string type binary variable.

For information, every time a character of our message is converted into its 8-bits binary characters, it should be stored in this variable and subsequently, the variable will always be used as input to print_bulb function when it is called by the main function to print out the colored emoji of 'on' and 'off' according to the input of 8-bits binary characters. Therefore, our goal is to get this representation of 8-bits binary character representation for every individual characters evaluated from the string of user's input message.

b) Thus, the code that gets the 8-bits binary characters representation for it to be sent to print_bulb function should be within the codeblock of our initial for loop statement that iterates every individual characters of our string message.

c) Furthermore, we need to first set the value of the last element of our string type binary variable equal to '\0'. This can be done by accessing the last index number of binary variable.

d) Next, as can be seen in the given hint # 2 earlier, we need to have another loop that every time it is looping, Hint #6: will divide a new variable that stores the initial value of ASCII decimal number of the relevant character, by 2 and the result of that division is updated back to that variable such that the condition to stop the loop is when the result of division eventually becomes zero value. Thus, Hint #7: perhaps, a while loop can be used for such operation and the condition of where the value of a new variable, not equal to zero, will ensure continuous iteration.

e) Within that while loop, apart from the division by 2 process and the updating of the value resulted from that division as described in d) above, Hint #8: a remainder operation using modulus operator shall also be performed as to initialize the string type binary variable the positional right-most binary character of either '1' or '0' value. That means, the first iteration will give a remainder for the right-most binary character, the second iteration, gives remainder for the second right-most and so forth. This process requires another variable that can keep track and monitor the index number of string type binary variable for proper placement of the binary characters accordingly. For example, right-most character will be associated to index number 7 of the string type binary variable, second right-most binary character is with index number 6, third right-most binary character is with index number 5 and so on. More importantly, this remainder is a numerical operation that its result gives a numerical value and as such, Hint #9: we need to have a conditional where if the remainder numerical is equal to 0, then its equivalent character '0' will be initialized to the string type binary variable as per its appropriate index number in the string. Vice versa, if remainder numerical value is 1, then its equivalent character of '1' will be initialized accordingly.

f) It should be noted also, and as has been shown in the example earlier where we may not get the full or complete 8-bits binary character representation. Thus, Hint #10: in our code, after the while loop and using the variable that keeps track the index number that the respective elements have been filled in with 0 and 1 character in string type binary variable due to remainder operation, we can have another condition to check whether the 8-bits binary character representation is complete or not. And if not, we need to prepend '0' to every unfilled element of string type binary variable using another loop, starting at the next right-most from the last index number registered in the previous while loop until its index number is zero. Once we have a complete series of 8-bits binary characters, the binary variable now for whatever evaluated character being iterated at the time, can be sent as input to print_bulb function for print out before the next character evaluation takes place.

3. COLORED EMOJIS

It has been mentioned before that the print_bulb function is partially completed and the incomplete part is only on the type of colored emoji to be used in the function as to print out the 'on' or '1' and 'off' or '0' based on the input of complete series of 8-bit binary characters.

In the print_bulb function, the printf statement of colored emoji is as follows:

```
printf("%s", /*TODO*/);
```

In the /*TODO*/, you are required to type in the unicode relevant to color emoji that is appropriate for 'on' and 'off' condition.

The unicode of colored emoji can be found at: https://emojiterra.com

For example, on that webpage, you can search the name of colored emoji relevant to your requirement based on your unique number. Once the emoji is found, scroll down until you see a table that shows the unicode for C/C++ & Python. For example, like below:

🟡 **Codes**

Shortcode (Discord)	**:yellow_circle:**
Shortcode (GitHub)	**:yellow_circle:**
Shortcode (Slack)	**:large_yellow_circle:**
HTML Dec	🟡
HTML Hex	🟡
CSS	\01F7E1
C, C++ & Python	\U0001f7e1
Java, JavaScript & JSON	\uD83D\uDFE1
Perl	\x{1F7E1}
PHP & Ruby	\u{1F7E1}
Punycode	xn--4g9h
URL Escape Code	%F0%9F%9F%A1

Thus, just copy or retype that relevant unicode at the second argument of your printf statement with double quotes in the print_bulb function, for example, printf("%s", "\U0001f7e1");

In the followings, are the color emoji requirements according to unique number:

Unique Number	On or '1'	'Off' or '0'
1	Red Heart	Black Heart
2	Orange Square	Green Square
3	Orange Circle	Green Circle
4	White Square	Purple Square
5	White Circle	Purple Circle
6	Red Square	Blue Square
7	Red Circle	Blue Circle
8	Yellow Square	Black Circle
9	Yellow Circle	Black Circle

FINAL ADVISES

1. Write pseudocodes as always so that you have a plan on the step-by-step instruction to get the correct out. Execute your plan as necessary and make changes in order to get the correct desired output. Comment your code necessarily.

2. Always test your code thoroughly to see if it works according the specifications or requirements of this assignment. In this part of assignment, you can use HI! example as a test case.

3. Once your code achieved the correct output, review your code's style and finally, the design aspects of it.

PART C - SAHIBBA

Alright, enough with secret coded messages and let's play word games!!!

BACKGROUNDS

In the simple version game of Sahibba, 2 players create words to score points, and the number of points is the sum of the point values of each letter in the word. The scoring for every alphabets are shown in **TABLE 1** as follows:

Table 1: Sahibba's scoring system

char	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
value	2	4	8	4	2	4	4	6	2	8	8	2	4	2	2	4	8	2	2	2	2	4	6	6	6	4

For example, if we wanted to calculate the score on word "Jambu", we would note that the 'J' is worth 8 points, the 'a' is worth 2 points, the 'm' is worth 4 points, the 'b' is worth 4 points and the 'u' us worth 2 points. Summing up these, we get that "Jambu" is worth 20 points.

However, here we have 3 different versions of Sahibba where each has its own major rule. Which is yours? That depends on your unique numbers and these are the version and their rules accordingly:

Unique number	Sahibba's Name	Major Rule:
1 - 3	SahibbaX	Word created must contain the alphabet chosen by one of the players.
4 - 6	SahibbaY	Word created must start with the alphabet chosen by one of the players.
7 - 9	SahibbaZ	Word created must contain minimum number of characters decided by one of the players.

Note that the rule is set through the command-line argument.

Other general rules applicable to all versions are as follows:

a) Only 1 word may be entered.
b) If word contains non-alphabetical characters, they are not counted for scores.
c) If the main rule and rule a) is violated, the player automatically receive 0 score or point.

HOW THE GAME WORKS

In general, when executing the code, alphabet in the case of SahibbaX / SahibbaY or number for Sahibba Z that becomes the rule is entered as command-line argument. For example, the second argument in the command-line argument is the rule:

```
$./sahibbaX B                (for Window's user: $.\sahibbaX.exe B)

$./sahibbaY M                (for Window's user: $.\sahibbaY.exe M)

$./sahibbaZ 5                (for Window's user: $.\sahibbaX.exe 5)
```

If the rule is valid, the rule will be announced.

Then, each player will be prompted to enter his/her own word. After that, their entered word will be checked against the rule. If the word entered complies with the main and general rule of a) and b) above, only then the word entered by player will be counted for points.

If both players do not comply with the rules, both players shall receive zero point and thus, both players are announced to be lost.

If only 1 player does not comply with the rules, the remaining player who complies with the rule shall have his/her points counted first and then will be declared as a winner by his winning point over zero score of the other disqualified player.

If both players receive same points or scores, they are announced as tie.

If one of the players has higher points, he/she will be announced as a winner.

The examples of code outputs are as follows:

1. SahibbaX

a) Example where both players are disqualified due to both words contain no alphabet set by the rule:

```
$./sahibbaX M
Rule - word must contain letter: M
Player 1: Sakan
Player 2: Kakak

Score:
Player 1: 0    Player 2: 0
BOTH PLAYERS LOSE!
```

b) Example where one of the players is disqualifed due to word entered consists of two words and the other player wins:

```
$./sahibbaX m
Rule - word must contain letter: M
Player 1: Makan Malam
Player 2: Jamban

Score:
Player 1: 0    Player 2: 22
PLAYER 2 WINS!
```

c) Example where both players scored same points and a tie is announced:

```
$./sahibbaX t
Rule - word must contain letter: T
Player 1: kotak
Player 2: KATAK

Score:
Player 1: 22    Player 2: 22
IT'S A TIE!
```

d) Example where one player wins due to higher score:

```
$./sahibbaX A
Rule - word must contain letter: A
Player 1: lapar
Player 2: laa

Score:
Player 1: 12    Player 2: 4
PLAYER 1 WINS!
```

e) Example where non-alphabetical characters are not counted for points:

```
$./sahibbaX n
Rule - word must contain letter: N
Player 1: Cinta...
Player 2: Cinta?

Score:
Player 1: 16    Player 2: 16
IT'S A TIE!
```

f) Example where uppercase and lowercase alphabetical characters are counted the same:

```
$./sahibbaX n
Rule - word must contain letter: N
Player 1: CINTA
Player 2: cinta

Score:
Player 1: 16    Player 2: 16
IT'S A TIE!
```

2. SahibbaY

a) Example where both players are disqualified due to both words contain no alphabet set by the rule:

```
$./sahibbaY M
Rule - word must start with letter: M
Player 1: Sampan
Player 2: Lambat

Score:
Player 1: 0    Player 2: 0
BOTH PLAYERS LOSE!
```

b) Example where one of the players is disqualifed due to word entered consists of two words and the other player wins:

```
$./sahibbaY m
Rule - word must start with letter: M
Player 1: Jumpa
Player 2: Minyak

Score:
Player 1: 0    Player 2: 24
PLAYER 2 WINS!
```

c) Example where both players scored same points and a tie is announced:

```
$./sahibbaY k
Rule - word must start with letter: K
Player 1: kotak
Player 2: KATAK

Score:
Player 1: 22    Player 2: 22
IT'S A TIE!
```

d) Example where one player wins due to higher score:

```
$./sahibbaY A
Rule - word must start with letter: A
Player 1: Andalusia
Player 2: Aman

Score:
Player 1: 20    Player 2: 10
PLAYER 1 WINS!
```

e) Example where non-alphabetical characters are not counted for points:

```
$./sahibbaY n
Rule - word must contain letter: N
Player 1: Nana?
Player 2: Nana!

Score:
Player 1: 8    Player 2: 8
IT'S A TIE!
```

e) Example where uppercase and lowercase alphabetical characters are counted the same:

```
$./sahibbaY n
Rule - word must contain letter: N
Player 1: NIKMAT
Player 2: nikmat

Score:
Player 1: 20    Player 2: 20
IT'S A TIE!
```

3. SahibbaZ

a) Example where both players are disqualified due to both words contain no alphabet set by the rule:

```
$./sahibbaZ 5
Rule - minimum number of letters in word: 5
Player 1: sini
Player 2: sana

Score:
Player 1: 0    Player 2: 0
BOTH PLAYERS LOSE!
```

b) Example where one of the players is disqualifed due to word entered consists of two words and the other player wins:

```
$./sahibbaZ 5
Rule - minimum number of letters in word: 5
Player 1: Sana
Player 2: Sungai

Score:
Player 1: 0    Player 2: 14
PLAYER 2 WINS!
```

c) Example where both players scored same points and a tie is announced:

```
$./sahibbaZ 5
Rule - minimum number of letters in word: 5
Player 1: kotak
Player 2: KATAK

Score:
Player 1: 22    Player 2: 22
IT'S A TIE!
```

e) Example where non-alphabetical characters are not counted for points:

```
$./sahibbaZ 3
Rule - minimum number of letters in word: 3
Player 1: Cinta?
Player 2: Cinta..!

Score:
Player 1: 16    Player 2: 16
IT'S A TIE!
```

e) Example where uppercase and lowercase alphabetical characters are counted the same:

```
$./sahibbaZ 6
Rule - minimum number of letters in word: 6
Player 1: NIKMAT
Player 2: nikmat

Score:
Player 1: 20    Player 2: 20
IT'S A TIE!
```

CODE SPECIFICATIONS

In this part of the assignment, we will find the three sahibba versions of .c program. In every one of them, the contains 2 global variables, where one of them is an integer array variable called POINTS and the other one is called RULE. Remember that global variables are variables declared outside of any functions including main function and they can be accessed by any functions.

The POINTS variable will be used to calculate the points of every players while RULE variable will take the value of second argument typed in the command-line argument if it is valid.

In all version of sahibba .c program, there are two functions created. One is the main function, and the other one, is a custom function to calculate points. Fortunately, our team leader has written pseudocodes in the main function. Basically, the main function will 1) read the rule, validate it and assign it to global variable rule if valid, 2) Prompt players to enter their words, 3) Compute and announce points earned by both players, and finally, 4) announce the winner.

Thanks to our team in Europe, they have completed the code that reads, validates and initializes the command-line argument to RULE global variable. Thus, it is the responsibility of your team to design and complete the codes in the main function and custom function that validates the words entered by both players and if valid, count the scores.

A) Design of main function

Since Europe's team has completed the first part of the code, which reads, validates and initializes argument in command-line argument to RULE global variable, we focus on the next pseudocodes:

i) Prompt both players to get their words: In this case, get_string function is used to prompt the players. For example, get_string("Player 1: ") with one whitespace after the colon. The same goes for player 2. Both their words will be stored in string variable of your chosen variable name, which will then be sent as inputs to custom compute_score function.

ii) Compute and announce scores: Here, the custom function of compute_score will be called with input arguments are the words entered by both players. The return scores, which are returned by the custom compute_scores function, are then initialized to some integer variables of your choice where announcement of the both players' scores will be made. The format of score announcement is as been shown in the examples earlier.

iii) Announce winner: In this part of the code, as has been shown in the example, there are four conditions to be coded. The first condition is when the scores of both players are zero, from which both players are announced as losers. The second condition is when both players score the same points, from which a tie announcement is made. The third and fourth condition is when one of the players win by highest points, for example, player 1 scores more point than player 2 and as a result, player 1 is announced as a winner.

B) Design of compute_scores function

The design of compute_scores function is based on the version of sahibba's game. However, in all version, the compute_scores is called twice where first, input of word entered by player 1 and second, input of word entered by player 2.

i) SahibbaX:

In SahibbaX, some initial value of score variable equals zero is initialized first. Then, we need to have a control variable to check wether alphabets in word entered complies with value of RULE global variable. To check for that, a for loop is employed where every individual character in word will be iterated since the main rule specifically stated that the word entered by player must contain the alphabet initialized in RULE. The limit to number of iteration can be found using the strlen function offered by string.h library. During the loop, if the word entered contains the required alphabet, then the value of control variable must be changed and once iteration is done, we need to check wether the value of control variable has changed from its initial initialized value or not. If the value remains the same as its initial value, that means the required alphabet of RULE global variable is not present in the entered

word, to which a return 0 statement can be employed returning 0 score to the main function.

During the loop or iteration, we can also check for whitespace since the presence of whitespace in between evaluated characters means that there are two words entered by player and this violated the rule of the game. Thus, a return 0 statement can be used here to return 0 score back to the main function.

Next, if the word entered by player is found to be valid, i.e, it is not against the rules, the individual characters in word is again, iterated one more time for counting scores. At this point, a condition must be employed to ensure only alphabetical characters and not non-alphabetical characters are considered for point scoring. Next, we convert every evaluated alphabetical character to uppercase letter using the toupper function offered by ctype.h libray. This ensures lowercase alphabets entered by players can earn the same point as the uppercase alphabets shown in TABLE 1.

To determine the points earned in every iteration, uppercase letter 'A' must be deducted from every evaluated alphabet. Remember that according to ASCII table, every uppercase alphabets has it equivalent integer decimal value and this value must be normalized accordingly. For example, uppercase letter 'A' has 65 as its ASCII decimal value and deducting 'A' from itself will normalize 'A' value to be 0. As for uppercase letter 'B', its ASCII value is 66 and by deducting 'A' from 'B' will normalize 'B' value to be 1. One more, uppercase 'Z' has its ASCII value of 90 and subtracting 'A' from 'Z' will normalize 'Z' value equal to 25. If this is done to every uppercase alphabet, the new normalize value for every uppercase alphabet can now be treated as the index number for POINTS array variable, and thus, earning the points or value specified in the POINTS array element defined by the index number.

For example, let's say that the current evaluated alphabet, which has been converted to uppercase letter is 'D' and its ASCII integer decimal number equals 68. If we subtract 'A' from 'D', that is we subtract 65 from 68, we will get 3 as the result of that subtracting operation. Then, using this number 3 as the index number for POINTS array variable, i.e. POINTS[3], this will leads us to the fourth element of POINT array variable and thus, earning the player with 4 points, which actually correspond to uppercase alphabet 'D' such as shown in Table 2 below.

TABLE 2: index number of POINTS array variable

char	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
value	2	4	8	4	2	4	4	6	2	8	8	2	4	2	2	4	8	2	2	2	2	4	6	6	6	4
Index no.	[0]	[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]

Above algorithm shall repeated in every iteration and every time points is earned according to the evaluated alphabet, those points will be added to score variable that is initially set to zero as has already been mentioned earlier. Once iteration is over, the total points in score is returned back to main function, in which it will be used for score announcement and also for determining the winner or winner announcement.

ii) SahibbaY:

In SahibbaY, some initial value of score variable equals zero is initialized first. Then, we need to check wether alphabets in word entered by player complies with the value initialized in RULE global variable. To check for that, only the first letter in word entered by player will be checked against the RULE global variable since the main rule specifically stated that the word entered by player must start with the alphabet initialized in RULE global variable. If the first alphabet is not equal to RULE, then we must use return 0 statement so that 0 score is returned back to the main function for that particular player's score.

If the word is found to be valid, we can now iterate over every individual character in word using for loop. The limit to number of iteration can be found using the strlen function offered by string.h library. By now we need to set a condition where only alphabetical characters and not non-alphabetical characters are considered for point scoring. Furthermore, we need to convert every evaluated alphabetical character to uppercase letters so that lowercase letter in word entered by player can receive the same points as uppercase letter shown in TABLE 1. To do this, we use toupper function offered by ctype.h library.

To get the point for every individual evaluated alphabetical characters, we need to deduct them with uppercase letter 'A'. This is to normalize the value of every uppercase alphabets evaluated in the loop. What does this mean? Remember that every uppercase alphabets has its equivalent integer decimal number as per ASCII table. For example, 'A' has ASCII value of 65 and if we deduct 'A' with 'A', then 'A' will have a normalize value equal 0. Let's do another one, 'B' has its ASCII value equal to 66 and by deducting 'A' from 'B", we get normalized value of 'B' equals 1. One last time, 'Z' has is ASCII value equals 90 and deduct 'A' from 'Z' gives us its normalized value as 25. Thus, this new normalized value can be treated as index number to POINTS array variable, which the value or points at particular index number can be earned.

For example, let's say that the current evaluated alphabet, which has been converted to uppercase letter is 'D' and its ASCII integer decimal number equals 68. If we subtract 'A' from 'D', that is we subtract 65 from 68, we will get 3 as the result of that subtracting operation. Then, using this number 3 as the index number for POINTS array variable, i.e. POINTS[3], this will leads us to the fourth element of POINT array variable and thus, earning the player with 4 points, which actually correspond to uppercase alphabet 'D' such as shown in Table 2 below.

TABLE 2: index number of POINTS array variable

char	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
value	2	4	8	4	2	4	4	6	2	8	8	2	4	2	2	4	8	2	2	2	2	4	6	6	6	4
Index no.	[0]	[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]

Above algorithm shall repeated in every iteration and every time points is earned according to the evaluated alphabet, those points will be added to score variable that is initially set to zero as has already been mentioned earlier. Once iteration is over, the total points in score is returned back to main function, in which it will be used for score announcement and also for determining the winner or winner announcement.

On final note, we haven't check for word entered by player containing more than 1 word. How to do this? During the iteration of every individual character in word entered by player, we can set another condition within the loop where if whitespace is found, a return 0 statement can be employed and thus, returning 0 score to the main function for that particular player score.

Iii) SahibbaZ

In SahibbaZ, some initial value of score variable equals zero is initialized first. Then, we need to check wether the number of alphabets in word entered by player complies with the value initialized in RULE global variable since the main rule specifically stated that the word entered by player must contain a minimum number of characters as per initialized in RULE global variable. To check for that, we need to iterate every individual character in the word. The limit to number of iteration can be found using the strlen function offered by string.h library. Inside the loop, we need to have a condition where only alphabetical characters are considered for checking the minimum number of character required by the rule. To count the number of eligible characters in the loop, we need to have a control variable, which every time alphabetical is found from the condition mentioned earlier, the control variable will be increased by 1 in every valid iteration. Once iteration is complete, the value of the control variable will be checked against the number initialized in the RULE global variable. If the value of control variable is less than the required number, this means that the minimum requirement is not met and as such, we need to employ return 0 statement where a 0 score is returned back to the main function for that player's score. If the minimum requirement is met, then we can proceed to the counting score.

During this checking iteration or loop, we can set another condition to see whether whitespace is present in the word entered by player. This means that instead of one required word, the player entered two words. Thus, we can again employ return 0 statement indicating that a 0 score is returned back to the main function for that player's score.

Now if the word entered by player meets all the requirement and not against any rule, we need to again, iterate the alphabetical characters for counting score. This time apart from the condition of only alphabetical character are considered for point scoring, we also need to convert the alphabetical characters to uppercase letters. This ensures the lowercase alphabets in the word entered by player receive the same points as the uppercase letters such as shown in TABLE 1.

To get the point for every individual evaluated alphabetical characters, we need to deduct them with uppercase letter 'A'. This is to normalize the value of every uppercase alphabets evaluated in the loop. What does this mean? Remember that every uppercase alphabets has its equivalent integer decimal number as per ASCII table. For example, 'A' has ASCII value of 65 and if we deduct 'A' with 'A', then 'A' will have a normalize value equal 0. Let's do another one, 'B' has its ASCII value equal to 66 and by deducting 'A' from 'B", we get normalized value of 'B' equals 1. One last time, 'Z' has is ASCII value equals 90 and deduct 'A' from 'Z' gives us its normalized value as 25. Thus, this new normalized value can be treated as index number to POINTS array variable, which the value or points at particular index number can be earned.

For example, let's say that the current evaluated alphabet, which has been converted to uppercase letter is 'D' and its ASCII integer decimal number equals 68. If we subtract 'A' from 'D', that is we subtract 65 from 68, we will get 3 as the result of that subtracting operation. Then, using this number 3 as the index number for POINTS array variable, i.e. POINTS[3], this will leads us to the fourth element of POINT array variable and thus, earning the player with 4 points, which actually correspond to uppercase alphabet 'D' such as shown in Table 2 below.

TABLE 2: index number of POINTS array variable

char	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
value	2	4	8	4	2	4	4	6	2	8	8	2	4	2	2	4	8	2	2	2	2	4	6	6	6	4
Index no.	[0]	[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]

Above algorithm shall repeated in every iteration and every time points is earned according to the evaluated alphabet, those points will be added to score variable that is initially set to zero as has already been mentioned earlier. Once iteration is over, the total points in score is returned back to main function, in which it will be used for score announcement and also for determining the winner or winner announcement.

- END -

Know when to get help,
do your best always,
and make us proud!