CSC 322 Systems Programming Fall 2025
Lab Assignment L1: Binary-handler
Due: Friday, September 19

1 Goal

In this first lab, we will develop a system program in C that processes both characters and numbers in their binary form. The program should be error-free and designed to allow the user to repeatedly execute its functions until they choose to exit. For students who are new to C, this project will serve an introduction to the language and its practical applications. For those who already have prior experience, it will provide a valuable opportunity to review fundamental concepts, strengthen programming skills, and prepare for the subsequence labs.

2 Introduction

A. Basic Commands

The system program provides four basic commands, as shown in `Table 1`. When a user enters a command, the program executes the corresponding function or exits, as illustrated in `Figure 1`. Note that each command needs parameters except for the command `exit`. If a command is misspelled, the program should handle it as an exception by displaying an appropriate error message. In such cases, the program must continue running and display the next prompt.

`Table 1. Basic Commands`

Command	Description
d2b	Convert decimal to binary. See the section II-B.
b2d	Convert binary to decimal. See the section II-B.
enc	Encrypt user's input based on Caesar's cipher. See the section II-C.
dec	Decrypt user's input based on Caesar's cipher. See the section II-C.
gcd	Encode a number into a gamma code. See the section II-D. (extra)
exit	Exit the program

The program **should run on a Linux machine at *cs.oswego.edu***. Once it is compiled and executed, it presents its own computing environment by displaying a prompt in the following format:

your_loginID $

`Figure 1` illustrates how the prompt appears when the **loginID** is *jwlee*. When a command is successfully executed, the corresponding output will be displayed immediately. The **loginID** should be printed by using the `printf` function.

```
jwlee@altair~ > cc lab1-jwlee.c -o lab1
jwlee@altair~ > ./lab1
jwlee $  exot
[Error] Please enter the command correctly!
jwlee $ d2b(7)
111
jwlee $ exit
jwlee@altair~ >
```

Figure 1. Command Mode

B. Conversion Between Binary and Decimal: `d2b/b2d`

The command **`d2b`** converts a given decimal number into its binary representation. Its syntax is as follows:

`d2b(argument1)`

The user's input should be provided in parentheses, as shown in `Figure 2`. Only decimal numbers are acceptable and characters including alphabets and special symbols such as comma (,), period (.), colon (:), semicolon (;) are not valid inputs. For incorrect command inputs, the program should display a simple error message.

```
jwlee $ d2b(10)
1010
jwlee $ d2b(101)
1100101
jwlee $ d2b(ten)
[Error] Please enter the command correctly!
jwlee $ d2b(10ten)
[Error] Please enter the command correctly!
jwlee $ dectobin(10)
[Error] Please enter the command correctly!
jwlee $
```

Figure 2. d2b example

The program must check and handle improper syntax, as shown in `Figure 2. d2b example`. Examples of improper syntax include:

- entering a misspelled command (e.g., `dectobin`)
- entering a non-numeric or character-based number (e.g., `d2b(ten)`)

After displaying the output or error message, the system must return to the prompt so the user can enter the next command.

The command **`b2d`** performs the conversion in the opposite direction: it converts a given binary number into decimal equivalent. Its syntax is as follows:

`b2d(argument1)`

The user's input should be provided in parentheses, as shown in **Error! Reference source not found.**. Only binary numbers consisting 0 and 1 are considered valid input. All other values - including decimal digits (2 to 9), alphabetic characters, or special symbols such as *comma* (,), *period* (.), *colon* (:), and *semicolon* (;) – must be treated as invalid. For incorrect command inputs, the program should display a simple error message.

```
jwlee $ b2d(10)
2
jwlee $ b2d(101)
5
jwlee $ b2d(ten)
[Error] Please enter the command correctly!
jwlee $ b2d(10ten)
[Error] Please enter the command correctly!
jwlee $ b2d(12)
[Error] Please enter the command correctly!
jwlee $ bintodec(10)
[Error] Please enter the command correctly!
jwlee $
```

Figure 3. b2d example

The program must check and handle improper syntax, as shown in the **Error! Reference source not found.**. Examples of improper syntax include:

- entering a misspelled command (e.g., `bintodec`)
- entering a non-numeric or character-based number (e.g., `b2d(ten)`)
- entering a non-binary number (e.g., `b2d(12)`)

After displaying the output or error message, the system must return to the prompt so the user can enter the next command.

C. Caesar's Cipher: `encrypt/decrypt`

Caesar's cipher is one of the oldest methods in cryptography, dating back to the time of Julius Caesar in the Roman Empire. He used this simple yet efficient technique to send and receive private messages with his subordinates and political allies. The method is a form of substitution cipher, in which each letter of the alphabet is replaced by another leeter according to a fixed rule. As shown in `Figure 4`, each letter in the plaintext is assigned a positional number: for example "A" is 1; "B" is 2, "C" is 3, and so on. To produce the ciphertext, each positional number is shifted to the right by a fixed amount. In `Figure 4`, the shift value is 3. According to this rule, "A" becomes the 4th character, "B" becomes the 5th character, and so forth.

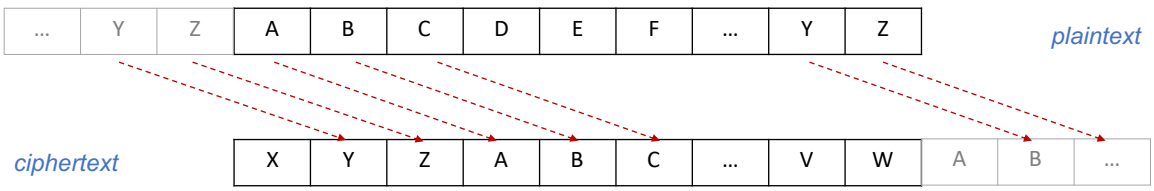

Figure 4. Shifts in Caesar's cipher method

The shifting rule presents an issue when applied to letters at the end of the alphabet such as Y and Z. For example, after a shift of 3, Y that was originally in 25th position would move to 28th position, which falls outside the range of the alphabet. Caesar solved this problem using modular arithmetic, meaning the shifted positional value is taken modulo the size of the alphabet (which is 26). For simplicity, let us assume *A* is in position 0, *B* is in position 1, and so on. Then the encryption formula $\mathbb{E}$ to obtain the ciphertext $\boldsymbol{C}$ from plaintext $\boldsymbol{P}$ is:

$$\boldsymbol{C} = \mathbb{E}(\boldsymbol{k}, \boldsymbol{P}) = (\boldsymbol{P} + \boldsymbol{k}) \, (\boldsymbol{mod}\ \mathbf{26})$$

In the formula, $\boldsymbol{k}$ represents the number of shifts (which is 3 in the earlier example). Similarly, decryption can be performed using the following formula $\mathbb{D}$:

$$\boldsymbol{P} = \mathbb{D}(\boldsymbol{k}, \boldsymbol{C}) = (\boldsymbol{C} - \boldsymbol{k}) \, (\boldsymbol{mod}\ \mathbf{26})$$

The ciphertext can be decrypted by only someone who knows the shift value $\boldsymbol{k}$. It makes the method secure. For example, the following sentence is encrypted to the ciphertext, which will be sent:

Actual message: `SEE ME AFTER CLASS`

Sending message: `VHH PH DIWHU FODVV`

Since the shift value $\boldsymbol{k}$ is given, the receiver can sassily decrypt the message back to the original plaintext. However, to those who do not know the value of $\boldsymbol{k}$, the message remains hidden.

However, Caesar's Cipher works only with the 26 letters of the English alphabet, meaning that numbers and special characters cannot be encrypted. In this program, you will extend the method to cover all characters in ASCII code set[1]. Each ASCII character is represented by an 8-bit number (traditionally using 7 bits). For example, "A" is 65, "B" is 66, the number 0 is 48, 1 is 49. Lowercases are also differentiated, for example, "a" is 97, "b" is 98. This also includes special characters such as "!", "+", and many others. By expanding encryption beyond the 26-letter alphabet, the level of security is significantly enhanced.

In your program, you must encrypt a plaintext to generate its ciphertext using $\boldsymbol{k} = 5$ (NOTE that $\boldsymbol{k}$ was 3 in the earlier examples). And you must decrypt a given ciphertext to recover the plaintext. The program will run as shown in `Figure 2`. It must accept the correct syntax:

enc(plaintext)
dec(ciphertext)

Any improperly entered commands should be detected and handled by displaying an appropriate error message, as illustrated in `Figure 2`.

The program must be able to check for the following error cases:

- entering a misspelled command, e.g., `encr(see you soon)`
- entering a command without parentheses, e.g., `enc "see you soon"`
- entering an incomplete command, e.g., `dec(rndb`

After executing a user's command, the system must return to the prompt so that the user can enter the next command. Are you ready? LTTI QZHP!!

```
jwlee $ enc(I have a key)
N%mf{j%f%pj~
jwlee $ enc (see me at 3)
xjj%rj%fy%8
jwlee $ dec (|jfw%ns%gqzj)
wear in blue
jwlee $ encr(see you soon)
[Error] Please enter the command correctly!
jwlee $ enc“see you soon”
[Error] Please enter the command correctly!
jwlee $ dec(rndb
[Error] Please enter the command correctly!
jwlee $
```

Figure 5. enc/dec example

D. [Extra Work - 6pts] Gamma Code: `gcd`

This command is optional, not mandatory. Students who implement it will earn up to 6 extra points. The **Gamma code** is designed to encode a decimal number into a special binary format using the following syntax:

gcd(decimal number)

The encoding process consists of two steps:

Step 1. Find the *offset* by cutting off the leftmost binary number 1 from the binary representation of the input number. For example, the decimal number 4 is 100 in binary, and the offset is 00, removing the leftmost 1. Note that 0 in the offset cannot be ignored.

Step 2: Encode the *length* of the offset using **Unary Code**[1]. In the sample above, the *offset* for 4 is 00, whose *length* is 2. The unary code of the *length* 2 is 110.

The **Gamma Code** is formed by concatenating the unary code for the offset *length* and the *offset* itself. Thus, the number 4 is encoded as 11000. Other examples are shown in `Table 2`. Compared to **Unary Code**, Gamma coding demonstrates better performance as the encoded number increases.

Table 2. Gammacode example

Number	Offset	Length	Length in unary code	Gammacode
1		0	**10**	0 (special case)
2	**0**	1	**10**	100
3	**1**	1	**10**	101
4	**00**	2	**110**	11000

[1] Unary Code encodes a number by printing a sequence of 1s equal to the number's value, followed by a single 0. For example, the number 2 is represented as 110, and the number 7 is encoded into 11111110 (seven 1s followed by 0).

9	**001**	3	**1110**	1110001
12	**100**	3	**1110**	1110100
20	**0100**	4	**11110**	111100100

The user's input must be provided in parentheses, as shown in `Figure 6`. Note that only decimal numbers are considered valid inputs. Any other characters including letters and special symbols such as *comma* (,), *period* (.), *colon* (:), *semicolon* (;) are not permitted. For incorrect command inputs, the program should display a simple error message.

```
jwlee $ gcd(2)
100
jwlee $ gcd(10)
1110010
jwlee $ gcd(ten)
[Error] Please enter the command correctly
jwlee $ gamma(10)
[Error] Please enter the command correctly
jwlee $
```

Figure 6. gammacode example

The program must check for and handle improper syntax, as shown in `Figure 6`. Examples of improper syntax includes:

- entering a misspelled command (e.g., `gamma`)
- entering a non-numeric or character-based input (e.g., `gcd(ten)`)

After displaying the output or error message, the system must return to the prompt so the user can enter the next command.

E. Exit Command: *exit*

The command has the following syntax:

exit

It must not have any arguments. If the input includes an argument or contains any typos, the program should detect the error and display an error message.

3 Hints in Implementation

A. Reading user input

There are several ways to receive user input in C, such as `scanf`, `sscanf`, `gets`, `fgets`, and so on. However, the best way to handle exceptions - such as non-numerical input or excessive input - is to read the input as a string and then parse it into the required forms. A good choice for this purpose is `fgets`, although you are not limited to using only this function in the lab. You can find full descriptions of these functions using the `man` command.

B. Parsing user input

Once the user enters a command, the program must parse it to determine the command type and its input argument. A tokenizer can be used to parse commands and return the information needed for further processing. The string library provides useful functions for this purpose, such as `strtok`. Full descriptions of the functions can be found using the `man` command.

4 Requirements

A. Developing environment

The program must be **implemented in C only**. Programs submitted in other languages will not be graded. The program must be executable on the CS Linux machine. Any program that cannot be compiled or executed on the CS Linux machine will be considered incomplete and will lose most of the points.

B. Handling exceptions

The program must detect errors while running. Errors may occur when user violates the command syntax or enters incorrect inputs (e.g., invalid characters, too many arguments, etc.; see Section II for more details). When an error is dected, the program must properly handle it by displaying an appropriate error message, while remaining runnable. **Programs that fail to detect or correctly handle errors will lose points**.

C. Readability of source code

The source code must be easy to read, with **clear indentation** and a **sufficient number of comments**.

D. Creating a readme file

The program must be submitted along with a README file that includes essential information such as your student ID, name, and any additional notes. Students completing extra work must notify the instructor through the README file. It may also contain special instructions for running the program, if applicable. The source file and README file must be named according to the following format. **Submissions that do not follow the naming convention will lose points**.

```
lab1-your_loginID.c
lab1-your_loginID_readme.txt
```

E. Submitting by due

The program must be submitted to Brightspace within the three weeks after the project is posted. **The due date is September 19**. Submissions made **by 11:59 PM** on the due date will be accepted without penalty. However, please note that Brightspace may experience heavy network traffic and instability on the due date. Therefore, you are strongly advised to submit earlier to avoid last-minute issues.

5 Grading

A. Grading criteria

The lab is worth **30** points, which accounts for 15% of the final grade. It will be graded based on how well the requirements are satisfied. Missing or unsatisfactory criteria will result in point deductions. The tentative grading breakdown is as follows:

- Compilation: **5** points
- Execution: **20** points
- Error detection & others: **5** points

Students who complete the extra work can earn up to **6** bonus points (equivalent to 20% additional credit for this lab).

B. Late penalty

Late submission will incur a **10% deduction per week** after the due date. **Submissions more than 10 weeks will not be accepted under any circumstances**.

6 Academic Integrity

Any dishonest behaviors will not be tolerated in this class. Plagiarism and cheating in any form will be addressed in accordance with the university's Academic Integrity Policy, available online at http://www.oswego.edu/integrity. For more information about university policies, see the following online catalog at:

http://catalog.oswego.edu/content.php?catoid=2&navoid=47#stat_inte_inte

Student who violate the honor code will receive no credits in this project.

[1] **ASCII code** with binary (not covering *UNICODE*)

Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char	Dec	Hex	Oct	Binary	Char
0	00	000	0000000	NUL (null character)	32	20	040	0100000	space	64	40	100	1000000	@	96	60	140	1100000	`
1	01	001	0000001	SOH (start of header)	33	21	041	0100001	!	65	41	101	1000001	A	97	61	141	1100001	a
2	02	002	0000010	STX (start of text)	34	22	042	0100010	"	66	42	102	1000010	B	98	62	142	1100010	b
3	03	003	0000011	ETX (end of text)	35	23	043	0100011	#	67	43	103	1000011	C	99	63	143	1100011	c
4	04	004	0000100	EOT (end of transmission)	36	24	044	0100100	$	68	44	104	1000100	D	100	64	144	1100100	d
5	05	005	0000101	ENQ (enquiry)	37	25	045	0100101	%	69	45	105	1000101	E	101	65	145	1100101	e
6	06	006	0000110	ACK (acknowledge)	38	26	046	0100110	&	70	46	106	1000110	F	102	66	146	1100110	f
7	07	007	0000111	BEL (bell (ring))	39	27	047	0100111	'	71	47	107	1000111	G	103	67	147	1100111	g
8	08	010	0001000	BS (backspace)	40	28	050	0101000	(	72	48	110	1001000	H	104	68	150	1101000	h
9	09	011	0001001	HT (horizontal tab)	41	29	051	0101001	)	73	49	111	1001001	I	105	69	151	1101001	i
10	0A	012	0001010	LF (line feed)	42	2A	052	0101010	*	74	4A	112	1001010	J	106	6A	152	1101010	j
11	0B	013	0001011	VT (vertical tab)	43	2B	053	0101011	+	75	4B	113	1001011	K	107	6B	153	1101011	k
12	0C	014	0001100	FF (form feed)	44	2C	054	0101100	,	76	4C	114	1001100	L	108	6C	154	1101100	l
13	0D	015	0001101	CR (carriage return)	45	2D	055	0101101	-	77	4D	115	1001101	M	109	6D	155	1101101	m
14	0E	016	0001110	SO (shift out)	46	2E	056	0101110	.	78	4E	116	1001110	N	110	6E	156	1101110	n
15	0F	017	0001111	SI (shift in)	47	2F	057	0101111	/	79	4F	117	1001111	O	111	6F	157	1101111	o
16	10	020	0010000	DLE (data link escape)	48	30	060	0110000	0	80	50	120	1010000	P	112	70	160	1110000	p
17	11	021	0010001	DC1 (device control 1)	49	31	061	0110001	1	81	51	121	1010001	Q	113	71	161	1110001	q
18	12	022	0010010	DC2 (device control 2)	50	32	062	0110010	2	82	52	122	1010010	R	114	72	162	1110010	r
19	13	023	0010011	DC3 (device control 3)	51	33	063	0110011	3	83	53	123	1010011	S	115	73	163	1110011	s
20	14	024	0010100	DC4 (device control 4)	52	34	064	0110100	4	84	54	124	1010100	T	116	74	164	1110100	t
21	15	025	0010101	NAK (negative acknowledge)	53	35	065	0110101	5	85	55	125	1010101	U	117	75	165	1110101	u
22	16	026	0010110	SYN (synchronize)	54	36	066	0110110	6	86	56	126	1010110	V	118	76	166	1110110	v
23	17	027	0010111	ETB (end transmission block)	55	37	067	0110111	7	87	57	127	1010111	W	119	77	167	1110111	w
24	18	030	0011000	CAN (cancel)	56	38	070	0111000	8	88	58	130	1011000	X	120	78	170	1111000	x
25	19	031	0011001	EM (end of medium)	57	39	071	0111001	9	89	59	131	1011001	Y	121	79	171	1111001	y
26	1A	032	0011010	SUB (substitute)	58	3A	072	0111010	:	90	5A	132	1011010	Z	122	7A	172	1111010	z
27	1B	033	0011011	ESC (escape)	59	3B	073	0111011	;	91	5B	133	1011011	[	123	7B	173	1111011	{
28	1C	034	0011100	FS (file separator)	60	3C	074	0111100	<	92	5C	134	1011100	\	124	7C	174	1111100	|
29	1D	035	0011101	GS (group separator)	61	3D	075	0111101	=	93	5D	135	1011101	]	125	7D	175	1111101	}
30	1E	036	0011110	RS (record separator)	62	3E	076	0111110	>	94	5E	136	1011110	^	126	7E	176	1111110	~
31	1F	037	0011111	US (unit separator)	63	3F	077	0111111	?	95	5F	137	1011111	_	127	7F	177	1111111	DEL