Base 64

Project Description

For this project, you will need to write C code to encode input data into "base 64" data. You will also need to write the C code to decode from "base 64" data back to its original values.

The concept of "base 64" encoding was invented to translate arbitrary lists of binary input bytes into printable ASCII characters, so they could be transmitted across networks. The basic idea is pretty simple. To encode arbitrary binary data, start with three bytes worth of data. Since each byte is 8 bits, that means we have 24 bits worth of data. Now divide these 24 bits into four 6 bit chunks. Each of these 6 bit chunks can be thought of as a number between 0 and 63, or a single "digit" in base 64. We then define a mapping for the numbers between 0 and 63 and a unique ASCII base 64 "digit". The resulting set of four ASCII characters is the base64 encoding of the original three bytes worth of data. The base 64 "digits" are defined in the following table:

Number	Bits	Symbol	Number	Bits	Symbol	Number	Bits	Symbol
0	000000	A	22	010110	W	44	101100	s
1	000001	B	23	010111	X	45	101101	t
2	000010	C	24	011000	Y	46	101110	u
3	000011	D	25	011001	Z	47	101111	v
4	000100	E	26	011010	a	48	110000	w
5	000101	F	27	011011	b	49	110001	x
6	000110	G	28	011100	c	50	110010	y
7	000111	H	29	011101	d	51	110011	z
8	001000	I	30	011110	e	52	110100	0
9	001001	J	31	011111	f	53	110101	1
10	001010	K	32	100000	g	54	110110	2
11	001011	L	33	100001	h	55	110111	3
12	001100	M	34	100010	i	56	111000	4
13	001101	N	35	100011	j	57	111001	5
14	001110	O	36	100100	k	58	111010	6
15	001111	P	37	100101	l	59	111011	7
16	010000	Q	38	100110	m	60	111100	8
17	010001	R	39	100111	n	61	111101	9
18	010010	S	40	101000	o	62	111110	+
19	010011	T	41	101001	p	63	111111	/
20	010100	U	42	101010	q	PAD	-	=
21	010101	V	43	101011	r			

For example, suppose we start with three input bytes of data, represented in ASCII as ‘A’ ‘B’ ‘C’, or, in hexadecimal, 0x41, 0x42, and 0x43. We can then encode these three bytes into base64 as follows:

Input	0x41								0x42								0x43							
Bits	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1
6bits	010000								010100								001001							
6bNum	16								20								9							
base64	Q								U								J							

Thus, the input values 0x41, 0x42, and 0x43 get encoded to the base64 string “QUJD”.

When the number of input bytes is not an exact multiple of 3, then we need to deal with a final set of input which is not completely specified. In this case, we translate as much data as we have to base64, padding to the right with zeroes to get six bit chunks. If a complete 6 bit chunk is missing, we pad the base64 with the padding character (an equal sign) so that the resulting base64 string is always an even multiple of 4 characters.

For example, if our last input string is only two bytes long instead of three, we would translate to base64 as follows:

Input	0x41								0x42															
Bits	0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	0								
6bits	010000								010100								001000							
6bNum	16								20								8							
base64	Q								U								I							

	--								=															

Thus, the input values 0x41, and 0x42 translate to the base64 string “QUI=”.

Similarly, if our last input string is only one byte long instead of three, we would encode to base64 as follows:

Input	0x41																							
Bits	0	1	0	0	0	0	0	1																
6bits	010000								010000								--							
6bNum	16								16															
base64	Q								Q								=							

	--								=															

Thus, the input value 0x41 translate to the base64 string “QQ==”.

Decoding base64 values are just the inverse of encoding. For a complete detailed description, see the Wikipedia article on base 64 at https://en.wikipedia.org/wiki/Base64.

Working on the Project

You have been provided with the basic infrastructure for the C code to perform base64 encoding and decoding. On the class web page, there is a file called proj1.tar.gz that you can download to your own UNIX directory. The command:

```
tar –xvzf proj1.tar.gz
```

will first create a sub-directory of your current directory called "proj1", and then populate that sub-directory with the contents of the proj1.tar.gz file. The proj1 sub-directory will contain the following files:

- encode.c – C source code that contains a "main" function that will read input, and pass that input to a function called "b64encode". Currently, the "b64encode" function does nothing… just returns an empty result. The main function takes the result and writes that data out.
- decode.c – C source code that contains a "main" function that will read input, and pass that input to a function called "b64decode". Currently, the "b64decode" function does nothing… just returns an empty result. The main function takes the result and writes that data out.
- base64.h – C source code that is already #include'd in encode.c and decode.c. This file contains declarations of a couple of variables that are associated with base64, and are designed to make your work easier.
- Makefile – a make file that contains build information for two executable files – encode and decode – as well as targets to test the results, clean the intermediate build and test files, and to create a final submission tar file.
- example.txt – an ASCII text file that contains some text that can be encoded to base64. This file is referenced in the Makefile test target.

Your job for this project is to write the b64encode function in encode.c, and the b64decode function in decode.c, and to test and debug those functions.

Here are some hints…

- Both b64encode and b64decode take two arguments. The first argument is an array of characters, and the second argument is the number of elements in that array. We haven't studied arrays yet, but you can access the j^{th} member of an array by using square brackets, like "string[j]", where j is some number between 0 and (len-1).
- Both b64encode and b64decode return a data type of "char *", which really is, under the covers, an array of characters. You may use global variables "b64Result" in b64encode, and "stringResult" in b64decode to hold the encoded or decoded (respectively) results. To write to the j^{th} member of this array, use square brackets in an assignment statement, such as "stringResult[j] = '=';", to put an equals sign into the j^{th} element of the stringResult array. In the result arrays, you may write to any element of the array from the zeroth element at stringResult[0] up to the last element at

stringResult[MAX_MSG_LEN-1]. Please add a byte with a zero value (a NULL terminator) after your last significant byte in either b64Result for b64encode, or stringResult in b64decode.

- A "b64Index" function is provided for you in base64.h. This function takes a base64 "digit" as a single ASCII character as an argument, and returns the index of that character (the "number" associated with that character) if the character is a valid base 64 digit. If the argument is not a valid base64 digit, then b64Index returns the number 64. Feel free to look at the code for this function, but it uses some concepts like standard library string functions and pointer arithmetic that we haven't studied, so as long as you understand what it does, don't worry too much about how it works. You will need this function in your b64decode implementation.
- Both the "encode" and the "decode" binary executable files read and write data using a UNIX feature called "stdin" and "stdout". By default, these are connected to the terminal and keyboard, so if you run encode, as in "./encode", then the terminal will open and wait for you to type input. When you type, your input is not read by the program until you hit either the "enter" key (which adds a newline to the input, sends the first line to your program, then opens the terminal for more input), or you hit the "Ctrl-d" key, which indicates "end of file". When you hit "Ctrl-d", the input is then sent to the b64encode function, and the output is written to the terminal.

  There is an alternative… use UNIX "redirection". Rather than reading from the terminal, you can add "<xyz.txt" to the command line to tell UNIX that you want to use the xyz.txt file for stdin rather than the terminal. You can also use ">abc.txt" to write the stdout output to file abc.txt rather than writing it to the screen. This makes it easy to run canned tests… "./encode <test1.txt >test1_b64.txt" will read the text in test1.txt, run your b64encode function on that data, and write the results to file test1_b64.txt. For more detail, see the Wikipedia article on standard streams at https://en.wikipedia.org/wiki/Standard_streams

Academic Honesty

You may find an implementation of the base64 algorithm in C code on-line. Please do not use this code. Your submission will be checked for plagiarism against both the web and other students' code.

However, you may look on the web for ideas and concepts that go in to making your own implementation of the project. If you do so, please include a comment in your code. e.g.

/* Concept from: https://xyz.code.org/base64 */

Submitting your Code

When you have finished coding and testing for this project, if you did not develop your code on an LDAP machine, please copy your code to an LDAP machine, and make sure it compiles and tests correctly in the LDAP environment. Then, on an LDAP machine, run:

```
make submit
```

in your proj1 directory. This will collect your source code and put it in a tar file called "<userid>_proj1.tar.gz", where "<userid>" is your LDAP user ID. (Note that your userid is encoded in tar file itself as well as in the file name, so just renaming the tar file is not good enough. You MUST run make submit on an LDAP machine.) Then upload the <userid>_proj1.tar.gz file onto blackboard in the Project 1 submission area.

Project 1 Grading

Project 1 is worth a total of 100 points. After the due date, your <userid>_proj1.tar.gz file will be downloaded onto an LDAP machine, untarred, and compiled using a Makefile similar to the one you have been using. If your code does not compile, you will get an 80 point deduction.

If your code compiles, but has warning messages, you will get a 10 point deduction for each type of warning message.

Your "encode" binary will then be run on the example.txt file provided to you. If your code does not produce the correct base64 encoding of example.txt, you will get a 20 point deduction.

Your "decode" binary will be run on a correct base64 encoding of example.txt. If your code does not reproduce example.txt, you will get a 20 point deduction.

Your "encode" binary will then be run on another sample text file that has not been published. If your code does not produce the correct base64 encoding of this sample, you will get a 15 point deduction.

Your "decode" binary will then be run on a correct encoding of the unpublished sample text. If your code does not reproduce the original sample, you will get a 15 point deduction.

Extra Credit

Expand the project to create two new C files … encipher.c and decipher.c. These should perform the same function as encode.c and decode.c, except after you read the input data, you should XOR each byte of input data with the character string specified as the first argument to encipher. (When you get to the end of the argument string, just go back to the beginning and continue XORing.) The decipher code should work similarly, but this time XOR the output of the b64decode function with the argument string before writing it to stdout.

This works because if you XOR data with any number, then XOR the result with the same number a second time, you get the original data back. After the first XOR, the data is scrambled in a fairly unpredictable way. (This is the basis for much computer security these days.) This is an implementation of a key based cipher, where the argument string is the "key". If you have the key, then you can decipher the message. But if you don't have the key, the message is very hard to decipher. XORing creates arbitrary, often unprintable, hexadecimal data… so we need the base64 encoding and decoding to make sure that our ciphered text is printable.

If you do the extra credit, add “encipher.c, decipher.c” to the “SRC_FILES” variable in the make file before running “make submit”.

If your encipher/decipher code works correctly on both the published and unpublished sample text, then you will get up to 20 points added to your grade. Even with this extra credit, your grade will not exceed 100 points.