CST8234 – C Programming W19 (Lab 6)

Programming Exercise

In this lab, we will gain experience with

- Binary File I/O
- Bit-wise operations

Background

Some data exchange protocols, most notable email, are not considered "8-bit safe". That means they expect all the content to be representable with just 7-bits. Clearly, this is a hold-over from the old days when all anyone cared about was ASCII.

But not being 8-bit safe not only impacts the ability to include non-English characters, it also prevents reliable transmission of attachments (e.g., images) that may contain any values.

Consequently, some protocols require that all data must be encoded to ensure it never uses the top bit in each bit. There are different protocol-specific standards for doing this. E.g., email uses "MIME encoding", some computers still rely on "uuencode" to transfer binary data between computers, and there is a general-purpose protocol called "Base 64".

All of the above-mentioned encoding schemes are fairly similar, so we'll just discuss Base64, which will give you an adequate grounding in the concept of encoding/decoding.

The general concept involves taking a stream of potentially binary data, recasting it as (slightly longer) string of **printable** characters. The ASCII table only has printable characters in the range of 32-126, i.e., there are just 95 possible printable characters to choose from. Meanwhile, each byte in our input data stream has 256 possible values.

So the basic approach is to not treat the binary data stream as a string of 8-bit bytes, but as a string of 6-bit "chunks", also known as a "sextet". In each 6-bit sextet, there can be $2^6 = 64$ possible values (hence the name "Base 64"). Because there are only 64 values, we can have enough printable characters in the ASCII table to safely represent each character.

Indeed one of the fundamental differences between Base 64, uuencode, and MIME based encoding is which printable character they map each of the 64 values to.

In the case of Base 64, it uses the ASCII values of the uppercase letters, the lower case letters, and the digits (in that order) which gives a total of 26+26+10 = 62 characters. To flesh out this list to a full 64 values, the characters '+' and '/' are defined as the last two values. See https://en.wikipedia.org/wiki/Base64#Base64_table for the lookup values.

Encoding

So to encode an input binary data stream, it is chopped into 6-bit sextets, each sextet is used as an index into our table of printable ASCII values (A-Za-z0-9+/). E.g., a value of 0 corresponds to 'A', a value of 3 corresponds to 'D', and value of 62 corresponds to '+'. These ASCII values are then streamed out as printable bytes.

E.g., a block of bytes such as 0x43, 0x53, 0x54, 0x38 … looks as follows in binary,

01000011 01010011 01010100 00111000 …

If treated as a continuous stream of bits and chopped into sextets, we get the following

010000110101001101010100001110 00 …

And each of those sextets (e.g., 010000=0x10=16, 110101=0x35=53, 001101=0x0e=13, 010100=0x14=20, etc.) are then looked up in the Base 64 table, and we find 16->'Q', 53->'1', 13->'N', 20->'U'

In other words, we turned the binary values 0x43, 0x53, 0x54 into the printable characters "Q1NU"

This process of chopping and looking up is repeated over and over on successive blocks of 3-bytes, each which are turned into 4 printable characters.

But because we are chopping our byte-based data stream into sextets (only 6 bits), and then are using those bits to choose which ASCII byte to stream out, we're somewhat inefficient. I.e., it's taking us 8 bits to represent 6-bits of binary data (but… our 8 bits are guaranteed to be printable characters!).

Consequently, each 3-byte block requires 4 printable characters (3 bytes * 8 bits per byte = 24 bits of data, 24 bits of data / 6 bits per sextet = 4 sextets, each then turned into a 8-bit printable character). This means that our resulting encoded string will be 4/3 = 1.333 times as long as the original. (Note: encoding is not to be confusing with compression. All compression algorithms encode data, but not all encoding algorithms compress data!)

The only tricky part of encoding is handling cases in which the input stream isn't divisible by three. In this case the last block of bytes may only contain one or two bytes and the binary data stream will need to be padded by the necessary number of zeros to bring it up to three, and then after looking up the first one or two sextets, one or two '=' characters are tacked on the end of the output stream to bring it up to 4 printable characters. See https://en.wikipedia.org/wiki/Base64#Examples. So you might get a Base 64 encoded stream that looks like

Q1NUODIzNA==

Indeed the time any padding '=' are tacked on, the length of resultant printable Base 64 encoded string is guaranteed to be evenly divisible by four.

Decoding

Once the remote server/program receives a Base 64 encoded block of data, it reverses the process. Each printable character (e.g., 'Q') is looked up in the Base 64 table to figure out which sextet would have generated that character (i.e., 'Q'->16, '1'->53, 'N'->13, 'u'->20. Successive sextets are then slapped together, so that four 6-bit sextets make 24 bits (i.e., three bytes) of binary data.

010000110101001101010100…

Which, when chopped into 8-bit bytes, gives us

01000011 01010011 01010100…

which is exactly the same as our original stream of binary bytes, i.e., 0x43, 0x53, 0x54.

Again, it's just the interpretation of the padding '=' at the end which is the only tricky bit, and that is well-illustrated in the Wikipedia article.

Your Task

You will be writing a Base 64 **encoder**, and a Base 64 **decoder**.

You have been given a makefile that contains the rules to build both by default.

You have also been given two files: base64_tables.c and base64_tables.h that set up the lookup tables, and provide the getters you'll need to do the lookups that will convert a sextet into a printable character (e.g., 16->'Q') and also a printable character into a sextet ('U'->20).

I've also provided two data files, haiku.txt and image.gif. One is actually a text file, but Base 64 doesn't care… bits are bits! Base 64 will happily encode either file.

You are required to write TWO programs. One will be implemented in a file called encode.c and one will be implemented in a file called decode.c. Both of those .c files will have their own main() function. The provided makefile will build two targets, each using only one of encode.c or decode.c. There is no need to split your code for each program across multiple files, as each of your programs will only be 50-100 lines in length.

Your encode program will take an optional file argument. If run without any arguments, your program will accept input from stdin. Alternatively, your program may be run with a single argument that is the name of a file that is to be used as the data source.

So in your program, you'll want to define a stream (e.g., "FILE* fileIn;"), and if argc is just 1, then simply set your stream to be stdio (literally, "fileIn = stdin;"). If argc is 2, then you'll actually have to do a fopen to open a file for **binary read** access. Of course, if there's a problem opening the file, you'll report it, and exit with a failure status.

This will allow you to invoke your program standalone, and type input into it (pressing CTRL-D to close stdin), or by providing a file name. The advantage of not using a file is that you can just "echo" input to

your program and play around with different length strings to make sure you're handling the padding correctly, as well as providing a content already entered into a file

```
$ echo -n "Hello, world!" | ./encode
SGVsbG8sIHdvcmxkAC==
$ ./encode haiku.txt
SW4gODIzNApNZW1vcnksIHBvaW50ZXJzIGFyZSBOVUxMClNlZ21lbnRhdGlvbiBmYXVsdCEKAA==
```

Your program will do successive reads of 3 bytes of the input stream, into a small buffer. Please check how to use "fread", as it will allow you to read up to three binary bytes, and will tell you how many it read. It is definitely the easiest way to approach this.

You will then convert each block of three bytes into four printable characters and print them to stdout. If you read less than 4 bytes (i.e., you're at the end of the input stream), then you'll appropriately pad (as per the description in the wikipedia article).

Of course, when your encode program exits, you must remember to close the stream if you opened a file.

Your decode program will also take an option file command line argument. This will determine whether you write your results to stdout (no command line argument), or to a file that you'll open for **binary write** access.

Your decode program will read its input stream (guaranteed to be text data!) from stdin.

You'll read successive blocks of 4 characters (here, doing an scanf(fileIn, "%3s",buf) will prove to be the simplest way) and convert them to blocks of 3 binary bytes, which you'll then dump to your output stream using "fwrite".

Testing

There are a ton of web-based Base 64 converters. E.g., https://www.motobit.com/util/base64-decoder-encoder.asp (picked at random) allows you to encode either files or strings. So you can check to make sure your encoder is working correctly by comparing your encoded string against the output of the same source data entered into an online converter.

Similarly, you can use an online convert to generate some encoded date, and then echo it into your decoder and you'll know your implementation is correct if generates the same source data that was used in the online converter.

Of course, you can also test your back-to-back implementation as follows

```
$ echo -n "Hello, world!" | ./encode | ./decode
Hello, world!
$ ./encode image.gif | ./decode image-out.gif
$
```

In the first example, if you've done everything correctly you'll clearly be able to see that you ended up with the same string as you input.

In the second example, you'll have to check in File Explorer to make sure that your output is the same (you can instantly tell by looking at the thumbnail for the file… it should be the same as image.gif)

But until you are certain that your encoder is perfect, you may want to use an online converter to generate the correct Base 64 encoding, and copy that encoded data into a test file that you can then echo to your decoder.

You are not required to write a usage function.

Gotchas

fread() requires a buffer. You can use a hard-coded array of chars, but don't forget to zero all the bytes before each fread.

scanf(fileIn, "%3s") requires a buffer. You can use a hard-coded array of characters, but don't forget to have one extra character for the NULL character that will get tacked on the end.

Signed/unsigned matters! Generally, you'll want to do everything as UNSIGNED, otherwise when you cast your byte from a char to a long, if the byte has a MSB of 1, then the extended 24 bits will also be '1'.

You will likely stumble across some bugs in your code relating to binary data. E.g., checking for null terminators, when 0 is a perfectly valid binary value.

You'll definitely want to test things carefully, making sure each block is correctly converted, and that padding characters are appropriately handled. I will not be testing with the same test files!

Requirements

1. Create a folder that uses either your Algonquin ID (e.g., "smit9112") or your both members' Algonquin ID's (e.g., "smit9112_sing0003"). Do all of your work in this folder, and when complete, submit the zipped folder as per the "Lab Instructions" posted on Brightspace.
2. Extract the files (.c, .h, makefile) in the lab attachment into your new folder. ***YOU ARE NOT REQUIRED TO MODIFY ANY OF THE CODE THAT YOU ARE BEING PROVIDED.***
3. Write a program in encode.c, and another program in decode.c that implement all the functionality described in this document. Your programs must be able to a back-to-back encode/decode of both text and binary data (this is why you are provided with two test files). If you decide to split your programs into multiple files, you'll need to edit the makefile appropriately.

Submission

When you are done, submit your program to Brightspace. Make sure that you have the appropriate header in your source file(s), and have zipped up the appropriately name directory. Only include the source code (.c, .h) ***including*** the files that were provided to you for this lab and your makefile (mandatory!). Do not include any other files (executables, stackdumps, vi "swap" files).

You ***must*** include a makefile, as part of your submission! When grading your reports I will unpack your zip file and type '`make`'… and if I don't end up with a 'lab5.exe' to run, ***I will not grade your assignment***.