How base64 works

Base64 is a encoding format encodes information from binary to text and vice-versa. The encoding has 64 characters defined and their respective binary representation. These 64 characters are chosen from a set of printable and common characters in many character encodings.

So since base64 wants to encode only 64 characters and we don't want to waste any unnecessary space, we can calculate how many bits we need:

$$\log_2(64) = 6$$

So we now know that we need only 6 bits to be able to be able encode all 64 characters, but most modern computers work with 8-bit bytes so to actually be able to encode this in a modern computer, we can find a multiple of 6 that is easier to work with on our system's byte size (assuming it's 8-bits).

This multiple is the LCM (lowest common multiple) between 6 and 8:

$$\mathrm{LCM}(6, 8) = 24 \text{ bits}$$

$$\frac{24}{8} = 3 \text{ bytes}$$

So we can walk through the binary data in 3 byte chunks to do the encoding and decoding.

The character table

Here’s the standard character table used by base64:

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

For padding we use the character =

Encoding

To start encoding I'll setup the lookup table in C so I can easily encode the 6 bits numbers.

```
static const uint8_t base64_encode_map[] = {
    [0] = 'A',  [1] = 'B',  [2] = 'C',  [3] = 'D',  [4] = 'E',  [5] = 'F',
    [6] = 'G',  [7] = 'H',  [8] = 'I',  [9] = 'J',  [10] = 'K', [11] = 'L',
    [12] = 'M', [13] = 'N', [14] = 'O', [15] = 'P', [16] = 'Q', [17] = 'R',
    [18] = 'S', [19] = 'T', [20] = 'U', [21] = 'V', [22] = 'W', [23] = 'X',
    [24] = 'Y', [25] = 'Z', [26] = 'a', [27] = 'b', [28] = 'c', [29] = 'd',
    [30] = 'e', [31] = 'f', [32] = 'g', [33] = 'h', [34] = 'i', [35] = 'j',
    [36] = 'k', [37] = 'l', [38] = 'm', [39] = 'n', [40] = 'o', [41] = 'p',
    [42] = 'q', [43] = 'r', [44] = 's', [45] = 't', [46] = 'u', [47] = 'v',
    [48] = 'w', [49] = 'x', [50] = 'y', [51] = 'z', [52] = '0', [53] = '1',
    [54] = '2', [55] = '3', [56] = '4', [57] = '5', [58] = '6', [59] = '7',
    [60] = '8', [61] = '9', [62] = '+', [63] = '/',
};
```

Now to start decoding we need to understand how we'll be doing the decoding.

Let's look at the 3 byte chunks we'll fetch at a time and how we'll need to extract the information:

$$\underbrace{111111}_{\text{part 1}}\underbrace{11 - 1111}_{\text{part 2}}\underbrace{1111 - 11}_{\text{part 3}}\underbrace{111111}_{\text{part 4}}$$

As you can see, the numbers overlap, so we'll need to do some bit shifting to extract the bits we care about. Let's do that.

$$\underbrace{111111}_{\text{part 1}}11$$

```
uint8_t part1 = (byte1 & 0b11111100) >> 2;
```

$$111111\underbrace{11 - 1111}_{\text{part 2}}$$

```
uint8_t part2 = ((byte1 & 0b11) << 4) | ((byte2 & 0b11110000) >> 4),
```

$$1111\underbrace{1111 - 11}_{\text{part 3}}$$

```
 uint8_t part3 = ((byte2 & 0b1111) << 2) | ((byte3 & 0b11000000) >> 6);
```

$$11\underbrace{111111}_{\text{part 4}}$$

```
uint8_t part4 = (byte3 & 0b111111);
```

After we extract all 4 parts from the 3 byte chunk, we can now encode it using the map we built.

```
uint_8 result[4] = {
  base64_encode_map[part1],
  base64_encode_map[part2],
  base64_encode_map[part3],
  base64_encode_map[part4],
};
```

We can essentially do this for every 3 byte chunks until we reach the end of the binary data we're encoding. But not all data out there is a multiple of 3. So we'll need to consider what happens when there's a remainder. So let's go:

```c
// data_size = how many bytes the data is
size_t remaining = data_size % 3;

if (remaining == 1) {
  uint8_t byte1 = data[data_size - 1];
  uint8_t part1 = (byte1 & 0b11111100) >> 2;
  uint8_t part2 = (byte1 & 0b11) << 4;

  result = (uint8_t[4]){
    base64_encode_map[part1],
    base64_encode_map[part2],
    // the = here is padding, it is optional as you can deduce it,
    // but I decided to comply with the standard.
    '=',
    '=',
  };
} else if (remaining == 2) {
  uint8_t byte1 = data[data_size - 2];
  uint8_t byte2 = data[data_size - 1];

  uint8_t part1 = (byte1 & 0b11111100) >> 2;
  uint8_t part2 = ((byte1 & 0b11) << 4) | ((byte2 & 0b11110000) >> 4);
  uint8_t part3 = ((byte2 & 0b1111) << 2);

  result = (uintn8_t[4]){
     base64_encode_map[part1],
     base64_encode_map[part2],
     base64_encode_map[part3],
     '=',
  };
}
```

Here's what's happening:

- when `remaining == 1`

$$\underbrace{111111}_{\text{part 1}}\ \underbrace{11}_{\text{part 2}}$$

Since there's only once byte remaining and the second half of part2 is on byte2, we can just discard it.

- when `remaining == 2`

$$\underbrace{111111}_{\text{part 1}}\underbrace{11 - 1111}_{\text{part 2}}\underbrace{1111}_{\text{part 3}}$$

Here the same logic applies, we discard the second part of part3 because it lies on byte3 and we only have 2 bytes left.

Decoding

The lookup table for decoding is just the reverse of what we had before:

```
static const uint8_t base64_decode_map[] = {
    ['A'] = 0,  ['B'] = 1,  ['C'] = 2,  ['D'] = 3,  ['E'] = 4,  ['F'] = 5,
    ['G'] = 6,  ['H'] = 7,  ['I'] = 8,  ['J'] = 9,  ['K'] = 10, ['L'] = 11,
    ['M'] = 12, ['N'] = 13, ['O'] = 14, ['P'] = 15, ['Q'] = 16, ['R'] = 17,
    ['S'] = 18, ['T'] = 19, ['U'] = 20, ['V'] = 21, ['W'] = 22, ['X'] = 23,
    ['Y'] = 24, ['Z'] = 25, ['a'] = 26, ['b'] = 27, ['c'] = 28, ['d'] = 29,
    ['e'] = 30, ['f'] = 31, ['g'] = 32, ['h'] = 33, ['i'] = 34, ['j'] = 35,
    ['k'] = 36, ['l'] = 37, ['m'] = 38, ['n'] = 39, ['o'] = 40, ['p'] = 41,
    ['q'] = 42, ['r'] = 43, ['s'] = 44, ['t'] = 45, ['u'] = 46, ['v'] = 47,
    ['w'] = 48, ['x'] = 49, ['y'] = 50, ['z'] = 51, ['0'] = 52, ['1'] = 53,
    ['2'] = 54, ['3'] = 55, ['4'] = 56, ['5'] = 57, ['6'] = 58, ['7'] = 59,
    ['8'] = 60, ['9'] = 61, ['+'] = 62, ['/'] = 63,
    // `=` -> padding
};
```

For decoding, we're assuming that all base64 strings we're given are already 3 bytes aligned, since we added = to pad it so it should always be used in 3 byte chunks.

First we decode the letter to their corresponding codes:

```
uint8_t part1 = base64_decode_map[base64_str[i]];
uint8_t part2 = base64_decode_map[base64_str[i + 1]];
uint8_t part3 = base64_decode_map[base64_str[i + 2]];
uint8_t part4 = base64_decode_map[base64_str[i + 3]];
```

Now the only thing we have to do is reverse the operations we did earlier.

```
uint8_t byte1 = (part1 << 2) | ((part2 & 0b110000) >> 4);
uint8_t byte2 = (part2 << 4) | ((part3 & 0b111100) >> 2);
uint8_t byte3 = (part3 << 6) | part4;
```

For more information check the wikipedia article