Merilyn Kuo
CSE 13S
Spring 2021

Assignment 2: A Small Numerical Library

Design Document

Purpose:

The purpose of this assignment is to learn about bits, nibbles, and bytes operations as well as parity bits by implementing a Hamming code encoder and decoder. We will be using knowledge from previous assignments such as abstract data types, matrices and arrays, parsing command line options, accessing files/input and output, loops, and if statements.

Pre-Lab Questions:

1. Lookup Table

0	0 (HAM_OK)		8	7
1	4		9	HAM_ERR
2	5		10	HAM_ERR
3	HAM_ERR		11	2
4	6		12	HAM_ERR
5	HAM_ERR		13	1
6	HAM_ERR		14	0
7	3		15	HAM_ERR

2. (a) Multiplying 11000111 with Ht, I get 1101. This is 1 on the lookup table so the error is at index one and we can flip that bit to get 11100001 in binary.

   (b) Multiplying 00011011 with Ht, I get 1011. This is 2 on the lookup table so the error is at index two and we can flip that bit to get 11011100 in binary.

Program Implementation:

Bit Vector ADT:

A bit vector can be represented as an array of ones and zeros. Since we're using Hamming(8, 4) codes for this assignment, the ADT will use an array of uint8_ts. The structure definition of the Bit Vector has been provided in the Assignment PDF. The structure members are the length in bits and the vector array of bytes.

```
struct BitVector {
    uint32_t length;
    uint8_t *vector;
}
```

*The pseudocode for this ADT was referenced from Sahiti's section.

The constructor for a bit vector allocates memory for the bit vector and initializes the members of the bit vector. If memory could not be allocated, then any allocated memory must be freed and the pointers nulled.

```
BitVector *bv_create(uint32_t length):
    BitVector *v = (BitVector *) calloc(sizeof(BitVector))
    if memory allocated:
        v->length = length
        v->vector = (uint8_t *) calloc(sizeof(uint8_t))
        if memory not allocated:
            free(v)
            v = null
    return v
```

The destructor function will free any allocated memory and set the vector pointer to null.

```
void bv_delete (BitVector **v):
    if v and v->vector:
        free(v->vector)
        free(v)
        *v = null
```

Return the length of the bit vector.

```
uint32_t bv_length(BitVector *v):
    return v->length
```

Sets the ith bit in the bit vector. The location of the byte is i / 8 and the position of the bit in that byte is i % 8.

```
void bv_set_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    v->vector[byte] or (1 << bit)
```

Clears the ith bit.

```
void bv_clr_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    v->vector[byte] and ~(1 << bit)
```

Returns the ith bit.

```
uint8 bv_get_bit(BitVector *v, uint32_t i):
    byte = i / 8
    bit = i % 8
    return (v->vector[byte] and (1 << bit)) >> bit
```

XORs the ith bit with the value specified.

```
void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit):
     byte = i / 8
     bit = i % 8
     v->vector[byte] xor (1 << bit)
```

Prints the bit vector.

```
void bv_print(BitVector *v):
    for i in range length
        print bv_get_bit(v, i)
```

Bit Matrix ADT:

The Bit Matrix ADT will use the Bit Vector ADT. It will help for performing matrix operations and encoding/decoding Hamming codes.

*The pseudocode for this ADT was referenced from Sahiti's section.

The constructor for a bit matrix allocates memory for the bit vector and initializes the members of the bit vector. If memory could not be allocated, then any allocated memory must be freed and the pointers nulled.

```
BitMatrix *bm_create(uint32_t rows, uint32_t cols):
    BitMatrix *m = (BitMatrix *) calloc(sizeof(BitMatrix))
    if m memory allocated success:
        m->rows = rows
        m->cols = cols
        m->vector = bv_create(rows * cols)
            if m->vector memory not allocated:
                free(m)
                *m = null
    return m
```

The destructor function will free any allocated memory and set the vector pointer to null.

```
void bm_delete(BitMatrix **m):
    if m and m->vector:
        free(m->vector)
        free(m)
        *m = null
```

Returns the rows in the bit matrix.

```
uint32_t bm_rows(BitMatrix *m):
    return m->rows
```

Returns the columns in the bit matrix.

```
uint32_t bm_cols(BitMatrix *m):
    return m->cols
```

Sets a bit in the bit matrix at a specified position.

```
void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t c):
    bv_set_bit(m->vector, r * (m->cols) + c)
```

Clears a bit in the bit matrix at a specified position.

```
void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c):
    bv_clr_bit(m->vector, r * (m->cols) + c)
```

Gets a bit in the bit matrix at the specified position.

```
void bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c):
    return bv_get_bit(m->vector, r * (m ->cols) + c)
```

Turns a byte of data into a bit matrix by creating a matrix and setting bytes or clearing bytes in the matrix.

```
BitMatrix *bm_from_data(uint8 byte, uint32_t length):
    BitMatrix *m = bm_create(1, length)
    for int i in range of length
        if byte and (1 << i)
            bm_set_bit(m->vector, i)
        else
            bm_clr_bit(m->vector, i)
    return m
```

Turns a bit matrix into data. ???

```
uint8_t bm_to_data(BitMatrix *m):
    uint8_t data = 0;
    for i, i < 8, i++
        bm_set_bit(m->vector, 1, i)
    return data
```

Performs matrix multiplication by multiplying each item in the rows of one matrix by each item in the columns of another. Returns the product.

```
BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B):
    BitMatrix C = bm_create(A->rows, B->cols)
    for i in A->rows
        for j in B->cols
            for k in A->cols
                bm_set_bit(C->vector, (bm_get_bit(i, k) *
                                       bm_get_bit(k, j)) % 2)
    return C
```

Prints the matrix.

```
void bm_print(BitMatrix *m):
    bv_print(m->vector)
```

Hamming Code Module:
Pseudocode referenced from Eugene's section.
Given a nibble of data (the lower nibble of msg), this function returns the Hamming code.

```
uint8_t ham_encode(BitMatrix *G, uint8_t msg):
   m = bm_from_data(msg, 4)
   c = bm_multiply(m, G)
   code = bm_to_data(c)
   return code
```

Given the Hamming code, this function decodes it with the transpose matrix and passes the decoded data back through the msg pointer. The decoded bits return the HAM_STATUS (enum definition given in Assignment PDF).

```
HAM_STATUS ham_decode(bitMatrix *Ht, uint8_t code, uint8_t *msg):
    c = bm_from_data(code, 8)
    error = bm_multiply(c, Ht)
    if error is 0
        no error
        return HAM_OK
    else
        return lookup(e)
    *msg = bm_to_data(error)
```

Encoding Program:
Based on the encoder program specifics and Eugene's section, the top level pseudocode for the main function will be as such:

```
Parse command line options with getopt loop
Generator Matrix = bm_create()
while fgetc() != EOF
    msg1 = lower nibble of byte read
    msg2 = higher nibble of byte read
    code1 = ham_encode(G, msg1)
    code2 = ham_encode(G, msg2)
    fputc(code1 and code2)
fclose(all files)
free(allocated memory)
```

For the encoding part, basically, I need to get data from the file, byte by byte. Each of the bytes are split into upper and lower nibbles that will be then encoded. After encoding, I can put them in the output file.

Decoding Program:
Based on the decoder program specifics and Eugene's section, the top level pseudocode for the main function will be:

```
Parse command line options with getopt loop
H transpose matrix = bm_create()
while fgetc() != EOF
    fgetc(code1)
    fgetc(code2)
    ham_decode(Ht, code1, *msg1)
    ham_decode(Ht, code1, *msg1)
    fputsc(pack_byte(msg1, msg2))
fprintf(statistics)
```

```
fclose(any open files)
free(allocated memory)
```

For the decoding part, essentially, I need to read each byte from the file and split it into a lower and upper nibble. Then I can use the decoding function to get the decoded byte. After, I put it back together and print it in the output file.

Credits and Resources Used:

Assignment PDF, Eugene’s Section and Sahiti’s Section.