Assignment 4
Hamming Codes

Prof. Darrell Long
CSE 13S – Winter 2021

Design Due: February 4th at 11:59 pm PST
Program Due: February 7th at 11:59 pm PST

1 Introduction

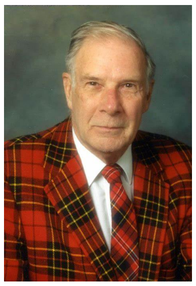
As we know, the world is far from perfect. In the communications domain, this imperfection is called *noise*. Noise (unwanted random disturbances) makes it difficult to have a reliable signal. Thus, transferring data through a noisy communication channel is prone to errors. Noisy channels are omnipresent. They are present in mobile phone networks and even the wires in a circuit. To counteract noisy interference, we add extra information to our data. This extra information allows us to perform error checking, and request that the sender retransmit any data that was incorrect. We can also add extra information to not only detect errors but also correct them. This technique is called *forward error correction* (FEC). CDs, DVDs, and even hard drives use FEC to account for scratches or bad sectors. In fact, most of our digital world such as Netflix would not be possible without FEC.

2 Hamming Codes

Hamming: *If you come to the Navy School, I will teach you how to be a great scientist.*
Long: *And if I go to Santa Cruz, will you still teach me?*
Hamming: *No.*

Lunch with Richard Hamming

Richard Wesley Hamming was an American applied mathematician whose work had many implications for computer engineering and telecommunications. His contributions include the Hamming code (which makes use of a Hamming matrix), the Hamming window, Hamming numbers, sphere-packing (or Hamming bound), and the Hamming distance. Hamming served as president of the Association for Computing Machinery from 1958 to 1960.


Richard Hamming

He used to joke that he was the *anti-Huffman*: David Huffman, the inventor of Huffman codes and a professor here at Santa Cruz, was a friend. Hamming added redundancy for reliability, while Huffman focused on removing it for efficiency.

In later life, Hamming became interested in teaching. From 1960 and 1976, before he left Bell labs, he held visiting positions at Stanford University, Stevens Institute of Technology, the City College of New York, the University of California at Irvine and Princeton University.

After retiring from the Bell Laboratories in 1976, Hamming took a position at the Naval Postgraduate School in Monterey, California, where he worked as an adjunct professor and senior lecturer in computer science, and devoted himself to teaching and writing books.

He spent significant effort in trying to recruit a young Dr. Long, telling him that "If you come to the Navy School, I will teach you how to be a great scientist." Dr. Long replied, "If I go to Santa Cruz, will you still teach me?" To which Hamming replied, simply, "No." Sadly, it was an opportunity lost.

2.1 Bits, Nibbles, and Bytes

Around 1948, John Wilder Turkey, an American mathematician, coined the word *bit* to replace the mouthful that was *binary digit.* A bit is the basic unit of information for digital systems. It represents a logical state with only two possible values, either a 1 or 0, on or off, true or false. But one bit is rather limiting. As a result, multiple bits were packed together to make up a *nibble* (4 bits with 2^4 states) or a *byte* (8 bits with 2^8 states).

2.2 Overview

Hamming codes are a linear error-correcting code invented by Richard W. Hamming [1] to correct errors caused by punched card readers. But, we will be using them to correct errors caused by random bit-flips (noise). He introduced the Hamming(7,4) code that encodes 4 bits of data D in 7 bits by adding 3 redundant or parity P bits. An explanation of parity bits will be given through an example. This code can detect and correct *one* error. With 7 possible errors, 3 parity bits can identify which bit is incorrect ($2^3 - 1 = 7$).

The Hamming(7,4) code is shown in table 1 where the Hamming code's least significant bit (LSb) is at index 001_2 and the most significant bit (MSb) is in position 111_2. While normally in practice we begin counting at 0, the importance of starting the count at 1 and in binary will be evident in the next paragraphs.

Index	111_2	110_2	101_2	100_2	011_2	010_2	001_2
Hamming code	D_3	D_2	D_1	P_2	D_0	P_1	P_0

Table 1: Hamming(7,4) code with data bits D and parity bits P.

You might notice that each parity bit's index has only one bit set (a power of 2). P_0's index (001_2) has the 0^{th} bit set, and you might notice that the index for D_0, D_1, and D_4 also have the 0^{th} bit set. Thus, P_0 is calculated over D_0, D_1, and D_4, *i.e.*, P_0 is set if the data bits have an odd number of 1s. This is an *even* parity scheme. With an odd parity scheme, the parity bit is set if by setting it there are an odd number of 1s. We could use an odd parity scheme, but for this assignment, we will be using an even parity.

We can calculate P_1 and P_2 in the same way where P_i is calculated over the data bits whose index has the i^{th} bit set. The formulas for the three parity bits P_i are shown below.

$$\begin{aligned} P_0 &= D_0 \oplus D_1 \oplus D_3 \\ P_1 &= D_0 \oplus D_2 \oplus D_3 \\ P_2 &= D_1 \oplus D_2 \oplus D_3 \end{aligned}$$

[1] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, April 1950.

As a result, each data bit has at least two parity bits that will help recover its value should an error occur. The overlap of parity bits also keeps them in check (parity bits can be erroneously flipped too). For example, the Hamming code for 0001_2 is shown in table 2. From here on, the index will follow convention and start at 0 and in decimal.

Index	6	5	4	3	2	1	0
Label	D_3	D_2	D_1	P_2	D_0	P_1	P_0
Hamming code	0	0	0	0	1	1	1

Table 2: Hamming(7,4) code for 0001_2.

where

$$\begin{aligned} P_0 &= 1 \oplus 0 \oplus 0 = 1 \\ P_1 &= 1 \oplus 0 \oplus 0 = 1 \\ P_2 &= 0 \oplus 0 \oplus 0 = 0 \end{aligned}$$

We currently have a *non-systematic* code, a code where the parity bits are placed throughout the code. While this is fine for a hardware implementation, it is tedious to do in software. Instead, we will be using a *systematic* code, a code where the parity bits are placed after the data bits. We will also be extending the Hamming(7,4) code by adding one more parity bit to make this a Hamming(8,4) code. The extra parity bit, P_3, is calculated over P_0–P_2 and D_0–D_3. If P_3 for a received code is incorrect then we know an error has occurred. Either P_3 is incorrect or one of the the bits in the code is incorrect. This extension has two additional benefits. First, each code is a byte rather than seven bits. Second, we can now detect two errors but only correct one. The new Hamming code is shown below.

Index	7	6	5	4	3	2	1	0
Hamming code	P_3	P_2	P_1	P_0	D_3	D_2	D_1	D_0

Table 3: Hamming(8,4) systematic code, an extension of the Hamming(7,4) code.

where

$$\begin{aligned} P_0 &= D_0 \oplus D_1 \oplus D_3 \\ P_1 &= D_0 \oplus D_2 \oplus D_3 \\ P_2 &= D_1 \oplus D_2 \oplus D_3 \\ P_3 &= D_0 \oplus D_1 \oplus D_2 \oplus D_3 \oplus P_0 \oplus P_1 \oplus P_2 \end{aligned}$$

2.3 Encoding

One approach to generating a Hamming code for a message is to calculate the parity bits one-by-one and appending them to the end of the message. Instead, we can use a generator matrix, $\boldsymbol{G}$. Given a message, $\vec{m}$, of four bits (a nibble) we can generate its hamming code, $\vec{c}$, by vector-matrix multiplication $\vec{c} = \vec{m}\boldsymbol{G}$

where the resulting code is eight bits in size (a byte). $\boldsymbol{G}$ is defined as:

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Vector-matrix multiplication for a vector $\vec{y}$ of size n and matrix $\boldsymbol{A}$ of size $n \times n$ where $\vec{y} = \vec{x}\boldsymbol{A}$ is defined as:

$$y_i = \sum_{k=1}^{n} x_k \cdot A_{i,k}.$$

Those of you with exposure to linear algebra (helpful but not required) will notice the left half of $\boldsymbol{G}$ is the identity matrix I (1s along the diagonal). This ensures the first four bits of the hamming code are the data bits while the right half is used to calculate the parity of message (notice the 0s along the diagonal). Generating the Hamming Code for $\vec{m} = \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$ (1100_2 in binary) is shown below. Note: binary is read from right to left with the MSb in the leftmost position and the LSb in the rightmost position. Vectors (arrays) are read from left to right.

$$\begin{aligned} \vec{c} &= \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \pmod{2} \\ &= \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 1 & 1 \end{pmatrix} \pmod{2} \\ &= \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{aligned}$$

In binary, $\vec{c}$ is equivalent to $1100\ 1100_2$.

Note: Addition and multiplication is mod2 in binary. You might remember from CSE 12 that $\bigoplus$ (exclusive-or) is the summing function and can be used in lieu of addition, and $\wedge$ (logical and) can substitute multiplication. These operations have the added benefit of operating in mod 2.

$$\begin{aligned} a \oplus b &= a + b \pmod{2} \\ a \wedge b &= a \times b \pmod{2} \end{aligned}$$

2.4 Decoding

The process for decoding a Hamming code is similar to encoding a message as it uses a parity-checker matrix, $\boldsymbol{H}$, to identify any errors and recover the original message if an error occurred. To decode a message, the code is multiplied by the transpose of the parity-checker matrix, $\vec{e} = \vec{c}\boldsymbol{H}^\top$ where $\vec{e}$ is the *error syndrome*, $\vec{c}$ is the Hamming code, and $\boldsymbol{H}$ is defined as

$$\boldsymbol{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

However, $\vec{e} \neq \vec{m}$. Instead, $\vec{e}$, the *error syndrome*, is a pattern of bits that identifies the error if there is one. For example, if $\vec{e} = [0,1,1,1]$, matching $\boldsymbol{H}$'s first column, then we know the error lies in the 0^{th} bit and flipping the 0^{th} bit in $\vec{c}$ will correct the error (remember the first four bits in the Hamming code are the data bits). If $\vec{e} = \mathbf{0}$ then our message does not contain an error. But, if the error syndrome is non-zero and the vector is not one of $\boldsymbol{H}$'s columns then we cannot correct the error since more than one bit has been flipped.

For example, to decode $\vec{c}$ calculated earlier, we can do the following:

$$\begin{aligned}\vec{e} = \vec{c}\boldsymbol{H}^\top &= \begin{array}{c}\begin{array}{cccccccc}0&1&2&3&4&5&6&7\end{array}\\ \begin{pmatrix}0&0&1&1&0&0&1&1\end{pmatrix}\end{array}\begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\\1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} \pmod 2\\ &= \begin{pmatrix}0&0&0&0\end{pmatrix}\end{aligned}$$

Since $\vec{e} = \begin{pmatrix}0&0&0&0\end{pmatrix}$, we know our message arrived with no errors. Since this is a systematic code, $\vec{m} = \begin{pmatrix}0&0&1&1\end{pmatrix}$ (the first four elements in $\vec{c}$).

If the second bit had been flipped due to noise and we received $\vec{c} = \begin{pmatrix}0&1&1&1&0&0&1&1\end{pmatrix}$ instead, we can calculate $\vec{e}$ to identify the flipped bit.

$$\begin{aligned}\vec{e} = \vec{c}\boldsymbol{H}^\top &= \begin{pmatrix}0&1&1&1&0&0&1&1\end{pmatrix}\begin{pmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\\1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} \pmod 2\\ &= \begin{pmatrix}1&0&1&1\end{pmatrix}\end{aligned}$$

Since $\vec{e} = \begin{pmatrix}1&0&1&1\end{pmatrix}$, $\boldsymbol{H}$'s second column or $\boldsymbol{H}^\top$'s second row, we know the second element in $\vec{c}$ was erroneously flipped. Flipping the value of the second element gives us $\vec{c} = \begin{pmatrix}0&0&1&1&0&0&1&1\end{pmatrix}$. As a result, $\vec{m} = \begin{pmatrix}0&0&1&1\end{pmatrix}$ or 1100_2, the original message. One possible method to determine which bit to flip is to compare the error syndrome with each column in $\boldsymbol{H}$, but section 2.5.5 will go over an optimal approach involving lookup tables.

2.5 Hamming Code Module

This module will implement the Hamming(8,4) code described in the earlier section. It's API is defined by `hamming.h` and as all provided files, you may not modify `hamming.h`. It's implementation should be written in `hamming.c`.

2.5.1 `ham_rc`

Most functions in this module will return a return code. The return code determines if a function failed (`HAM_ERR`) or was successful (`HAM_OK`). In the case of decoding, an error can be recoverable (`HAM_ERR_OK`). The return codes supported by the module are shown below and are provided in `hamming.h`.

ham_rc

```c
typedef enum ham_rc {
    HAM_ERR    = -1,
    HAM_OK     = 0,
    HAM_ERR_OK = 1
} ham_rc;
```

2.5.2 `ham_rc ham_init(void)`

This should be called before any function in the module as it creates and initializes the ***G*** and ***H*** Bit Matrix ADTs. If creating either of them fails, it must return `HAM_ERR` or `HAM_OK` otherwise.

2.5.3 `void ham_rc ham_destroy(void)`

To prevent any memory leaks, the Hamming module must free any memory it allocated. To check for memory leaks, you must use `valgrind`. Your program must pass valgrind cleanly.

2.5.4 `ham_rc ham_encode(uint8_t data, uint8_t *code)`

To generate a Hamming code for a nibble of data, we pass a byte with the nibble in the lower half in `data`. Providing a pointer to `code` allows us to return a return code while still updating `code` with the Hamming code. If the Hamming code was successfully generated, it will return `HAM_OK` and the value in `code` is valid. In the case the module has not been initialized or the `data`, `code` pointers are invalid, it must return `HAM_ERR`.

2.5.5 `ham_rc ham_decode(uint8_t code, uint8_t *data)`

To decode a Hamming code, we need to provide the code and a location to store the decoded data. If decoding was successful and did not require any correction it should return `HAM_OK`, and value in `data` will be valid. But if correction was required and successful, it should return `HAM_ERR_OK` and `data` is updated with the original message. But, in the case that the module was not initialized, `data` is not a valid pointer, or the error could not be corrected, it should return `HAM_ERR` and the value pointed to by `data` should be ignored.

To avoiding comparing the error syndrome with each column in ***H*** until a match is found (or no match is found and the error cannot be corrected), we can refer to a lookup table. A lookup table is an array that contains precomputed information that is referred to often. By constructing a lookup table, we can avoid performing the same computation many times at the expense of storing the table in memory (in this case, storing 2^4 or 16 bytes is *negligible*). The index to the table will be the error syndrome and the value is the bit(start counting at 0) that needs to be flipped if the error can be corrected. Thus, if $\vec{e} = \begin{pmatrix}1 & 0 & 1 & 1\end{pmatrix}$ or 1101_2 (13_{10}) then `table[13]` = 1. If the error cannot be correct then `table[`$\vec{e}$`] = HAM_ERR`.

Pre-lab Part 1

⚠ Pre-labs are part of the design document, and answers to the questions should be at the head of the document.

1. Calculate Hamming codes for 0000_2–1111_2 using the generator matrix. Show your work. *Hint*: Convert your codes to hex.
2. Decode the following codes. If it contains an error, explain how you can correct it; however, some errors cannot be corrected.
   (a) $1110\,0011_2$
   (b) $1101\,1000_2$
3. Complete the rest of the look-up table shown below.

0	0
1	4
…	…
15	HAM_ERR

3 Bit Matrix

Bit Matrices, a variation of the bit vector, are a rarely taught but essential tool in the kit of all computer scientists and engineers. A Bit Matrix is an ADT that represents a two dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0). For this assignment, the generator matrix $\boldsymbol{G}$ and parity-checker matrix $\boldsymbol{P}$ will be implemented as a Bit Matrix.

This is an efficient ADT since, in order to represent the truth or falsity of a matrix of $n \times n$ items, we can use $\lfloor n/8 \rfloor + 1$ `uint8_t`s per row instead of n, and being able to access 8 indices with a single integer access is extremely cost efficient. Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte. The bit matrix is defined as

Bit Matrix struct

```
struct BitMatrix {
    uint32_t rows;
    uint32_t cols;
    uint8_t **mat;
};
```

3.1 `BitMatrix *bm_create(uint32_t rows, uint32_t cols)`

This constructor will allocate memory for the `BitMatrix` ADT and the matrix. The number of elements in the matrix is rows × cols. If at any point allocating memory with `calloc()` fails, the function must return `NULL`, else it must return a `BitMatrix *` or a pointer to a `BitMatrix`.

3.2 `void bm_delete(BitMatrix **m)`

The destructor will free the memory allocated for the `BitMatrix` ADT. This requires freeing memory for the bit matrix followed by freeing the structure itself. It is best practice to set a pointer to `NULL` once it has been freed to avoid dereferencing a `NULL` pointer in the future (in reality you should check if a pointer is valid before dereferencing it). In order to set `m` to `NULL`, we need the address of the pointer to the `BitMatrix` ADT or a double pointer.

Freeing Memory

```c
// Incorrect
void free_memory(uint8_t *arr) {
    free(arr);

    // The value of arr is passed by value
    // so only the copy of arr is set to NULL.
    arr = NULL;
}

// Correct
void free_memory(uint8_t **arr) {
    // Need to dereference once as free()
    // expects a void * as an input
    free(*arr);

    // This will set the value of arr outside
    // of this scope to NULL.
    *arr = NULL
}
```

3.3 `uint32_t bm_rows(BitMatrix *m)`

This is the first of two accessor functions. This will return a `BitMatrix`'s number of rows.

3.4 `uint32_t bm_cols(BitMatrix *m)`

This will return a `BitMatrix`'s number of cols.

3.5 `void bm_set_bit(BitMatrix *m, uint32_t row, uint32_t col)`

Upon creation, the elements of the matrix will be initialized to 0. However, to construct the generator and parity-checker matrix, certain elements of the matrices must be set. To set a bit in the matrix, it is necessary to access the byte where the bit specified by `row` and `col` is located. The location of the byte is given by the following: mat[row, $\lfloor$ col /8 $\rfloor$]. Then bitwise operations are needed to set the bit. Setting a bit should not interfere with any of the other bits' value.

3.6 `void bm_clr_bit(BitMatrix *m, uint32_t row, uint32_t col)`

In some cases it may be necessary to clear a bit or element in the matrix. Similarly to setting a bit, it is necessary to access the byte where the bit specified by `row` and `col` is located. The location of the byte is given by the following: `mat[row, `$\lfloor col/8 \rfloor$`]`. Then bitwise operations are needed to clear the bit. Clearing a bit should not interfere with any of the other bits' value.

3.7 `uint8_t bm_get_bit(BitMatrix *m, uint32_t row, uint32_t col)`

When performing vector-matrix multiplication, we will need the value of the bit at the specified `row`, `col`. To get a bit's value in the matrix, it is necessary to access the byte where the bit specified by `row` and `col` is located. The location of the byte is given by the following: `mat[row, `$\lfloor col/8 \rfloor$`]`. Then bitwise operations are needed to get the value of the bit. This function should return a 1 if the bit is set or 0 otherwise.

3.8 `void bm_print(BitMatrix *m)`

For debugging purposes it is helpful to print the `BitMatrixrix`. *Do this first.*

4 Your Task

For this assignment, you will be implementing a Hamming code library that uses the Hamming(8,4) code explained earlier. To implement ***G*** and ***H***, you will be using a bit matrix ADT.

G* and *H

```
// Both G and H are 4x8 bit matrices.
BitMatrix *generator;
BitMatrix *parity;
```

You will also be creating two small programs for this assignment. One for generating Hamming codes and another for decoding Hamming codes. They must read from `stdin` by default or a file and write to `stdout` by default *or a file*. Thus, they should support command-line arguments, `-i` and `-o`, to specify input and/or output files. The decoder will also print statistics such as total bytes processed, uncorrected errors, corrected errors, and the error rate to `stderr`.

We will be providing source code for a program (`error.c`) that will inject errors (noise) into your Hamming codes. Note that not all errors will be correctable, but your output should match ours. The rate at which the program injects errors is a command-line argument and is specified by `-e rate` (default is

```
$ ./gen -i frankenstein.txt | ./dec | diff frankenstein.txt -
  Total bytes processed: 902092
  Uncorrected errors: 0
  Corrected errors: 0
  Error rate: 0.000000
```

Figure 1: Example usage and decoder statistics. Here `diff` uses the dash to represent `stdin`.

0.01 or 1%) and must be between [0.0, 1.0]. The seed is specified with `-s seed` and it must be a positive integer.

```
$ ./gen -i frankenstein.txt | ./err -e 0.002 -s 2021 | ./dec -o out.txt
  Total bytes processed: 902092
  Uncorrected errors: 89
  Corrected errors: 14267
  Error rate: 0.000099
```

Figure 2: Example usage with added noise

4.1 File I/O and Permissions

In the last assignment, the grid was read from `stdin` or a file, and the final grid was printed to `stdout` or a file. Thus, you should be familiar with file I/O. However, we must pay special attention to the input and output file's permissions when generating or decoding Hamming codes. For example, imagine you have a sensitive file on your UNIX machine that only you can read or write. Everyone else will receive an error if they attempt to read or write to it. The file containing the generated Hamming codes should also only be read or written by you, *i.e.*, it should inherit the file permissions of the file for which it is generating Hamming codes. This will prevent another user with prying eyes to decode the file with Hamming codes and learn the contents of your sensitive file (if it is *that* sensitive, it should be encrypted in the first place).

4.1.1 UNIX File Permissions

Every file in UNIX has a set of access modes for the owner, group, and everyone else. The `ls -l` command will lists all the files in a directory and also displays its permissions. In figure 3, the owner of the file, `bat-notes`, is batman, and the group is `dc`. Users in UNIX can be members of a group such as the `sudo` group. The `groups` command lists the groups the current user is in. Groups allow users to share files within their groups while controlling the extent of their access.

The file's permissions are presented in the same order from left to right after the first dash: owner, group, and others. Read permission is represented by `r`, write permission, the ability to modify, create, or delete a file, is represented by `w`, and execute permission for programs and shell scripts are represented by `x`. Thus, in figure 3, batman has the following permissions on his file, bat-notes: `rw-rw-r-`. The

```
$ ls -l
  -rw-rw-r-- 1 batman dc 102 Jan 21 12:33 bat-notes
```

Figure 3: Output of `ls -l`

owner, `batman`, has read and write permissions but cannot execute the file. The group, `dc`, has read and write permissions but cannot execute the file. Everyone else can only read the file.

File permissions are not set in stone and can be modified by `chmod` (both a command and a system call). For example, if batman wants to remove read access to everyone else other than himself and members of the dc group, he can execute the command `chmod o-r bat-notes` to remove read permissions from others. If he changes his mind later, he can use the command `chmod o+r bat-notes` to add read permissions to others.

```
$ chmod o-r,g-w bat-notes
$ ls -l
  -rw-r----- 1 batman dc 102 Jan 21 12:33 bat-notes
```

Figure 4: Removing other's read permission and the group's write permission

As mentioned earlier, `chmod` is also a system call and can be called from a **C** program. For this assignment, all output files should inherit the file permissions of the input file. `fstat()` should be used to retrieve the input file's permissions and `fchmod()` to set the output file's permissions to match the input's. Refer to `man chmod, man fchmod, man fstat` for more information. Note: both of these functions expect a *file descriptor* that is returned by low-level IO (`open()`), so you will need to use `fileno()` to get the file descriptor of an open stream.

Using `fstat()` and `fchmod()`

```c
FILE *in = fopen("bat-notes", "r");
// Hamming codes for "bat-notes"
FILE *out = fopen("bat-notes-HC", "w");
struct stat buf;

// Getting and setting file permissions
fstat(fileno(in), &buf);
fchmod(fileno(out), buf.st_mode);
```

Another way to represent file permissions is with three *octal* digits, one for owner, group, and others. The octable table is shown in table 4. In fact, those with a keen eye will notice that file permissions are in fact a set. Thus, if a file has its permissions set to 755_8 then the owner has read, write, and execute permissions and the group and others only have read, execute permissions.

Ref.	Octal	Binary
`- - x`	1	001
`- w -`	2	010
`- w x`	3	011
`r - -`	4	100
`r - x`	5	101
`r w -`	6	110
`r w x`	7	111

Table 4: Octal and binary representation of file permissions.

5 Specifics

This assignment will require *two* programs, one to generate Hamming codes and another to decode them. Your `Makefile` should also build a third program *err*. The source code for the error generator will be provided in the resources repository.

Helper functions

```c
// Returns the lower nibble of val
uint8_t lower_nibble(uint8_t val) {
    return val & 0xF;
}

// Returns the upper nibble of val
uint8_t upper_nibble(uint8_t val) {
    return val >> 4;
}

// Packs two nibbles into a byte
uint8_t pack_byte(uint8_t upper, uint8_t lower) {
    return (upper << 4) | (lower & 0xF);
}
```

5.1 Generator program

1. Parse the command-line options with `getopt()` and open any input and/or output files.
2. Initialize the Hamming Code module with `ham_init()`.
3. Read a byte from the specified file stream or `stdin` with `fgetc()`.

4. Generate the Hamming(8,4) codes for both the upper and lower nibble with `ham_encode()` and write to the specified file or `stdout` with `fputc()`. The Hamming code for the lower nibble should be written first followed by the code for the upper nibble. Note: You'll notice this operation becomes repetitive with larger files. If only there was a lookup table of Hamming codes to refer to.
5. Repeat steps 3–4 until all data has been read from the file or `stdout`.
6. Use `ham_destroy()` to free memory allocated by the Hamming Code module.
7. Close both the input and output files with `fclose()`.

Note: If an input and output file are specified, the output file should have the same file permissions as the input file. You can use `fstat()` to retrieve an open file's permissions, `fchmod()` to change the permissions of the output file to match the input's. Since both of these functions expect a *file descriptor* that is returned by low-level IO (`open()`), you will need to use `fileno()` to get the file descriptor of an open stream.

5.2 Decoder program

1. Parse the command-line options with `getopt()`.
2. Initialize the Hamming Code module with `ham_init()`.
3. Read *two* byte from the specified file stream or `stdin` with `fgetc()`. Note: The first byte read is the Hamming code for the lower nibble, and the second is the upper nibble.
4. For each byte pair read, decode the Hamming(8,4) codes for both with `ham_decode()` to recover the original upper and lower nibbles of the message. Then, reconstruct the original byte. Note: The return code from `ham_decode()` should be used for statistics. Your program should count the number of bytes proccessed, Hamming codes that required correction, and Hamming codes that could not be corrected.
5. Write the reconstructed byte with `fputc()`.
6. Repeat steps 3–5 until all data has been read from the file or `stdout`.
7. Call `ham_destroy()` to free memory allocated by the Hamming Code module.
8. Print the following statistics to `stderr` with `fprintf()`:
    - Total bytes processed: The number of bytes read by the decoder.
    - Uncorrected errors: The number of Hamming codes that could not be corrected.
    - Corrected errors: The number of Hamming codes that experienced an error that was recoverable.
    - Error rate: The rate of uncorrected errors for a given input. This can be calculated by the following formula: $e = \frac{u}{t}$ where u is the number of uncorrected errors, and t is the total number of bytes (Hamming codes) processed.
9. Close both the input and output files with `fclose()`.

Note: If an input and output file are specified, the output file should have the same file permissions as the input file. You can use `fstat()` to retrieve an open file's permissions, `fchmod()` to change the permissions of the output file to match the input's. Since both of these functions expect a *file descriptor* that is returned by low-level IO (`open()`), you will need to use `fileno()` to get the file descriptor of an open stream.

6 Deliverables

You will need to turn in:

1. `generator.c`: This file will contain your implementation of the Hamming Code generator and should support the following command-line arguments:
   - `-i input_file`
   - `-o output_file`
2. `decoder.c`: This file will contain your implementation of the Hamming Code decoder and should support the following command-line arguments:
   - `-i input_file`
   - `-o output_file`

   It should print the following statistics to `stderr`:
   - Total bytes processed: The number of bytes (Hamming codes) read by the decoder.
   - Uncorrected errors: The number of Hamming codes that could not be corrected.
   - Corrected errors: The number of Hamming codes that were corrected.
   - Error rate: The rate of uncorrected errors for a given input. This can be calculated by the following formula: $e = \frac{u}{t}$ where u is the number of uncorrected errors, and t is the total number of bytes (Hamming codes) processed.
3. `bm.h`: This fill will contain the bit matrix ADT interface. This file will be provided. You *must* not modify this file.
4. `bm.c`: This file will contain your implementation of the bit matrix ADT. You *must* define the bit matrix struct here and not in bm.h.
5. `hamming.h`: This file will contain the interface of the Hamming Code module. This file will be provided. You *must* not modify this file.
6. `hamming.c`: This file will contain your implementation of the Hamming Code module.
7. `error.c`: This file will be provided in the resources repository, but should be included in your repository. You *must not* modify this file.
8. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. At this point you will have learned about `make` and can create your own `Makefile`.

- Your Hamming code generator executable must be named `gen`, and `make gen` should build it.
- Your hamming code decoder executable must be named *dec*, and `make dec` should build it.
- The error generator executable must be named *err*, and `make err` should build it.
- `make` should build your programs, as should `make all`.
- `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
- `CC=clang` must be specified.
- `make clean` must remove all files that are compiler generated.
- `make format` should format any source or header files with the course-supplied `clang-format` file.

9. `README.md`: This must be in *Markdown*. This must describe how to build and run your program.

10. `DESIGN.pdf`: This *must* be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. **C** code is **not** considered pseudocode. You *must* push `DESIGN.pdf` *before* you push *any* code.

7 Submission

Calvin: Hocus-pocus abracadabra! I command my homework to do itself! Homework, be done! Rats.

Bill Watterson, *Calvin and Hobbes*

To submit your assignment through `git`, refer to the steps shown in `asgn0` Remember: *add, commit,* and *push*! Your assignment is turned in *only* after you have pushed and submitted the commit ID to Canvas. Your design document is turned in *only* after you have pushed and submitted the commit ID to canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

8 Supplemental Readings

The more that you read, the more things you will know. The more that you learn, the more places you'll go.

—Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 2 §2.9
  - Chapter 5 §5.7

    - Chapter 7
- Hamming Codes