Hamming Code Implementation in C++

Introduction

Hamming code is an error detection/correction technique developed by Richard Hamming a mathematician in the early 20th century, the technique focuses on correcting one-bit errors and detecting if there are more than one, it's based on the parity checker, where there are multiple parity bits that detect if there's an even parity among the 1's in a particular order.

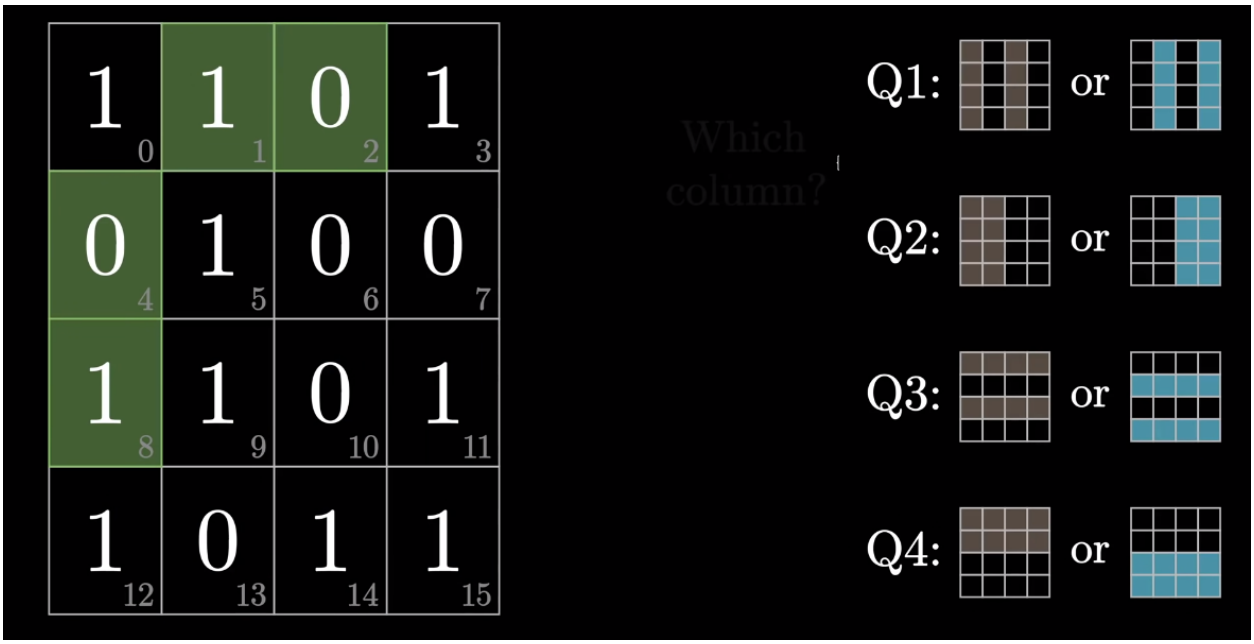


Figure 1. patterns for determining parity bits at 1, 2, 4, 8 in a 15-11 hamming code

courtesy of [3blue1brown]

The parity bits as seen in figure 1 are distributed in order of powers of 2, this feature makes it an efficient way of detecting the bits, since the bits in powers of 2 can make the orders highlighted in the blue in figure 1, there are 4 orders for checking for a 16-bit hamming code, and there is only 10 for a 1MB hamming code, which is only 0.001% from the total data sent. Although this is an efficient way of sending and detecting data, it's limited to only correcting 1 bit.

C++ code structure and functionality

The program written in C++ follows a number of steps, firstly there are multiple text files which will represent the transmitter and receiver data

- the file at the transmitter side called `Original Message` which holds the message to be transmitted
- the file Error Message which holds the corrupted data that was received
- the file Hamming Code Message is the file that holds the message coded in hamming code
- the file Corrected Hamming Code which holds the corrected hamming code after being corrupted
- the file Corrected Message which holds the restored message after being decoded

```
std::ifstream file(R"(C:\Users\Hp\Desktop\Original Message.txt)");
std::ofstream ErrorMessageFile(R"(C:\Users\Hp\Desktop\Error Message.txt)",
std::ios::trunc);
std::fstream CorrectHammingMessageFile(R"(C:\Users\Hp\Desktop\Hamming Code
Message.txt)");
std::fstream ErrorHammingMessageFile(R"(C:\Users\Hp\Desktop\Hamming Code Message
Error.txt)");
std::fstream CorrectedHammingFile(R"(C:\Users\Hp\Desktop\Corrected Hamming
Code.txt)");
std::ofstream CorrectedMessageFile(R"(C:\Users\Hp\Desktop\Corrected Message.txt)",
std::ios::trunc);
```

after opening the listed files, the message at the receiver begins to be read and encoded to hamming code when executing the while loop below. let's for example say the message is "Hello!".

```
while (file.read(&byte, 1)) {
    countBytes++;
    std::vector<int> HammingCodedByte;
    HammingCodedByte.reserve(20);
    HammingCodedByte = HammingEncoder(byte);
    //write encoded hamming code to file as integers
    int HammingCodeInteger = ConvertBinaryVectorToInt(HammingCodedByte);
    CorrectHammingMessageFile << HammingCodeInteger << "\n";
    std::vector<int> HammingCodeWithError;
    HammingCodeWithError.reserve(20);

    HammingCodeWithError = generateRandomError(HammingCodedByte);
    //print hamming code with error to error file
    HammingCodeInteger = ConvertBinaryVectorToInt(HammingCodeWithError);
    ErrorHammingMessageFile << HammingCodeInteger << "\n";
    char errorByte = HammingDecoder(HammingCodeWithError);
    ErrorMessageFile.put(errorByte);

}
```

in the while loop, the text file is read byte-wise until the EOF, in this loop the vector HammingCodeByte holds the encoded hamming code of the said byte, generated by the function HammingEncoder, then, the vector (holding the bits of hamming code one bit in each index) is converted to an integer and it is written on the file CorrectHammingMessage for the purpose of comparison, this file will not be used later.

Another vector HammingCodeWithError is defined and set to a vector holding a corrupted hamming code, generated by the function generateRandomError, the corrupted hamming code is then written to a file as an integer for correction in the next snippet.

After that the corrupted char is written to another file ErrorMessageFile

```
    while(ErrorHammingMessageFile >> HammingValue){
        std::vector<int> CorrectedHammingCode;
        CorrectedHammingCode.reserve(20);
        CorrectedHammingCode = HammingCodeCorrector(HammingValue);
        HammingValue = ConvertBinaryVectorToInt(CorrectedHammingCode);
        CorrectedHammingFile << HammingValue << "\n";
        char CorrectedChar = HammingDecoder(CorrectedHammingCode);
        CorrectedMessageFile.put(CorrectedChar);
    }
}
```

now for correcting the ErrorMessageFile that has corrupted data, the corrupted hamming codes are read from the file with corrupted hamming data.

To achieve this, the function HammingCodeCorrector returns the hamming code to its original form, then it gets decoded and written as a character.

Code functions

There are a handful of functions defined to help achieve the functionality of the code

1. HammingCodeEncoder

```
std::vector<int> HammingEncoder(char byte) {

    std::vector<int>HammingCode;
    HammingCode.reserve(20);
    int shmt = -1;
    for (int i = 0; i <= 12; ++i) {
        if(isNotPowerOf2(i)){
            shmt++;
            int bit = (byte >> shmt) & 1;
            HammingCode.push_back(bit);
        }
        else{
            HammingCode.push_back(0);
        }
    }
    //now to fill the bits in the 2^n indexes
    //1. shift the vector indexes by a shmt := 0
    //  a. test for ___1
    //  b. test for __1_
    //  c. test for _1__
    //  d. test for 1___
    //  where shmt(shift amount) is changed each iteration for the 4 cases
    //2. mask by 1
    //3. xor it with the previous bit
    //4. update the parity indexes
    int i = 0;
    shmt = 0;
    int XORValue = 0;
    while((1 << i) <= HammingCode.size()){//loop until index is larger than the
size of the vector
        int index = 1 << i;// to fill the indexes with multiple of 2
        for (int j = 0; j < HammingCode.size(); ++j) {
            if(isNotPowerOf2(j)){
                if((j >> shmt) & 1){//if the index is not a power of 2 and has a 1
at place n:
                    XORValue = XORValue ^ HammingCode[j];

                }
            }
        }
        HammingCode[index] = XORValue;
        XORValue = 0;
        shmt++;
        i++;

    }

    return HammingCode;
}
```

in this function the first *for* loop is for filling the vector and skipping the indexes with powers of 2 which will be preserved for the parity bits and initializes them with zeros.

the *while* loop begins to fill the parity bits based on xoring the bits in the orders denoted in figure 1.

The first order is an xoring between the indexes that have 1 as their LSB, which are 3, 5, 7, 9 and 11. the second order are the indexes which have the configuration XX1X as their bits, X being 0 or 1, these indexes are 3, 6, 7, 10 and 11.

Notice that we keep shifting to the right and check if the condition is true or not, if it is true (i.e. the index has 1 at the bit position n) then we need to xor that index and store the result of xoring in the corresponding parity bit.

Note that 1 << i is the same as 2^i.

2. generateRandomError

```
 std::vector<int> generateRandomError(std::vector<int> HammingCode) {
    count++;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(1, 12);

    int randomValue = distr(gen);
    HammingCode[randomValue] = !HammingCode[randomValue];
    return HammingCode;
}
```

this function generates a random integer between 1 and 12, and then flips the bit of the corresponding hamming code vector, then returns the vector with the corrupted bit.

3. HammingDecoder

```
    char HammingDecoder(std::vector<int> HammingCode) {
    int RecoveredByte;
    //the bits holding the info are those that are not a multiple of powers of 2
    int ParityIndex = 1;
    int i = 0;
    HammingCode[0] = 9;
    while (ParityIndex <= HammingCode.size()){
        HammingCode[ParityIndex] = 9;
        ParityIndex = (1 << (++i));
    }
    HammingCode.erase(std::remove(HammingCode.begin(), HammingCode.end(), 9),
HammingCode.end());
    for (int j = 0; j < HammingCode.size(); ++j) {
        RecoveredByte += HammingCode[j] * (1 << j);
    }


    return (char)RecoveredByte;
}
```

this function takes a hamming code vector as a parameter, the vector contains 8 bit of the 12 bits as data, the parity bits are not part of the data. This function replaces the parity bits with the integer value of 9.

Then the .erase built in function is used in conjunction with the std::remove function with the value 9 as the value that need to get removed. This leaves the hamming code vector with only the data of the byte.

After recovering the bits of the byte read, they are converted back to a char.

4. ConvertBinaryVectorToInteger

```
int ConvertBinaryVectorToInt(std::vector<int> HammingCode) {
    int HammingInteger = 0;
    for (int j = 0; j < HammingCode.size(); ++j) {
        HammingInteger += HammingCode[j] * (1 << j);
    }

    return HammingInteger;
}
```

this function takes a vector with bits as it's data, and combines them into a single decimal that is returned after.

5. HammingCodeCorrector

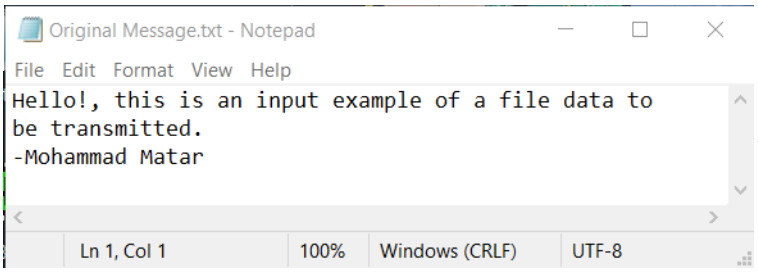
```
std::vector<int> HammingCodeCorrector(int value) {
    std::vector<int> CorrectedHammingCode;
    CorrectedHammingCode.reserve(20);
    int shmt = -1;
    for (int i = 0; i <= 12; ++i) {

        shmt++;
        //int shift = value >> shmt;
        int bit = (value >> shmt) & 1;
        CorrectedHammingCode.push_back(bit);
    }

    int XORValue = 0;
    int i = 0;
    shmt = 0;
    int index = 0;
    while((1 << i) <= CorrectedHammingCode.size()){//loop until index is larger
than the size of the vector
        for (int j = 0; j < CorrectedHammingCode.size(); ++j) {

            if((j >> shmt) & 1){//if the index is not a power of 2 and has a 1 at
place n:
                XORValue = XORValue ^ CorrectedHammingCode[j];

            }
        }
        index += XORValue * (1 << i);
        XORValue = 0;
        shmt++;
        i++;
    }
    if(index != 0){
        CorrectedHammingCode[index] = ! CorrectedHammingCode[index];
    }
    return CorrectedHammingCode;
}
```

this function takes the integers stored in the file Hamming code with error, stores the bits of each integer in a vector, then that vector is tested for all the orders listed in figure 1, if there is an error it will be at the index that is the result of concatenating the xor values of each order, if the concatenation result is 0, there will be no errors, if not, the error is detected and corrected at that index.

Program Results and Output

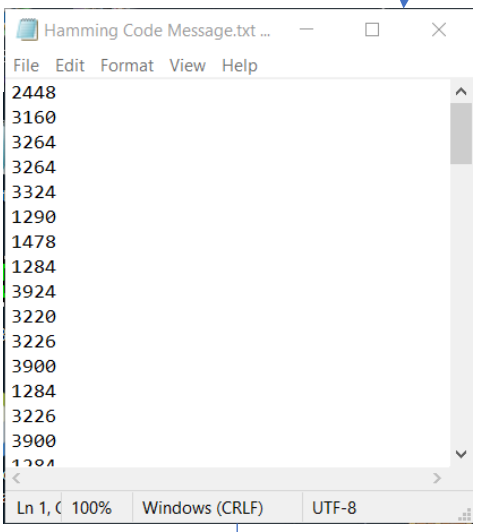
After executing the program, the files listed will look like this:

Figure 2. data getting ready to be transmitted



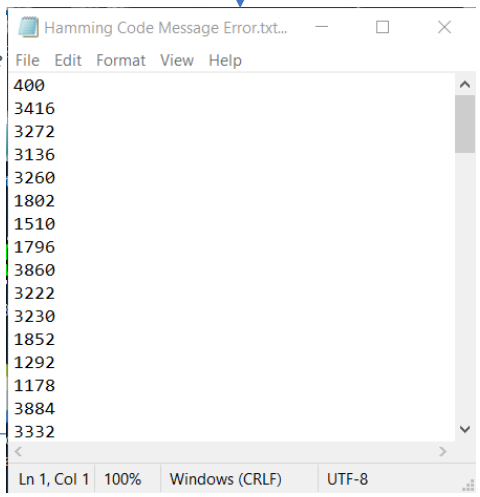
Encoding the message

Figure 3. encoded message.



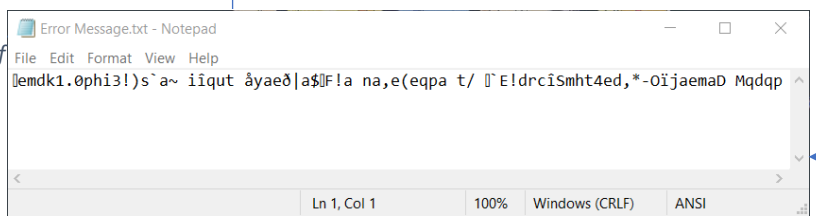
Introducing random error

Figure 4. Hamming code with error.



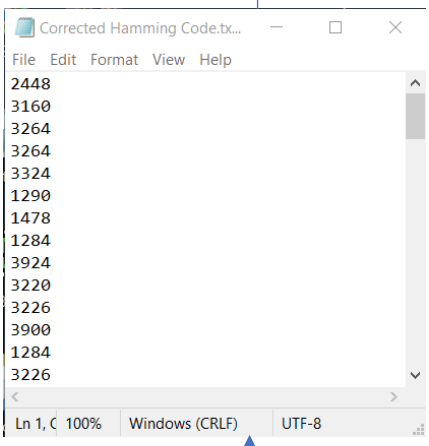
Decoding the corrupted hamming code

Figure 5. message of corrupted hamming code.



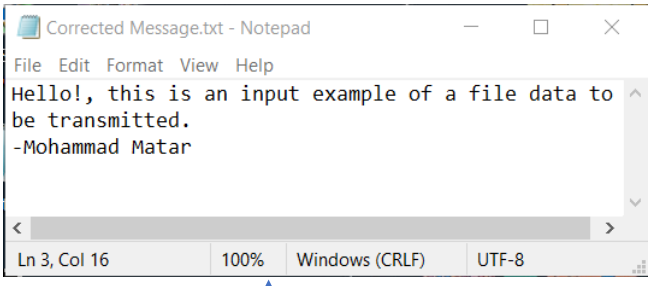
Correcting the corrupted code

Figure 6. corrected hamming code.



Decoding the corrected message

Figure 7. recovered message



Conclusion

Hamming code is a powerful and efficient tool for error detection, and although the program written for it may not be the best implementation since a 1MB would take about 13 seconds to encode, generate a random error and decode.

Hamming code would be great for hardware implementation since it only depends on the XOR logic.

Appendix

In this appendix all of the program's code will be displayed.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>
int countBytes = 0;
std::vector<int> HammingEncoder(char byte);

std::vector<int> generateRandomError(std::vector<int> HammingCode);

char HammingDecoder(std::vector<int> HammingCode);
int ConvertBinaryVectorToInt(std::vector<int> HammingCode);
std::vector<int> HammingCodeCorrector(int value);
int count;
int main() {
    // Open the file in binary mode

    std::ifstream file(R"(C:\Users\Hp\Desktop\temp.txt)");
    std::ofstream ErrorMessageFile(R"(C:\Users\Hp\Desktop\Error Message.txt)",
std::ios::trunc);
    std::fstream CorrectHammingMessageFile(R"(C:\Users\Hp\Desktop\Hamming Code
Message.txt)");
    std::fstream ErrorHammingMessageFile(R"(C:\Users\Hp\Desktop\Hamming Code Message
Error.txt)");
    std::fstream CorrectedHammingFile(R"(C:\Users\Hp\Desktop\Corrected Hamming
Code.txt)");
    std::ofstream CorrectedMessageFile(R"(C:\Users\Hp\Desktop\Corrected Message.txt)",
std::ios::trunc);

    // Check if the file is opened successfully
    if (!file.is_open() || !ErrorMessageFile.is_open()) {
        std::cerr << "Error opening files." << std::endl;
        return 1;
    }
    if (!CorrectHammingMessageFile.is_open()) {
        std::cerr << "Error opening files." << std::endl;
        return 1;
    }
    if (!CorrectedHammingFile.is_open()) {
        std::cerr << "Error opening files." << std::endl;
        return 1;
    }

    if (!ErrorHammingMessageFile.is_open()) {
```

```
        std::cerr << "Error opening files." << std::endl;
        return 1;
    }

    // Read the file byte-wise until the end
    char byte;
    while (file.read(&byte, 1)) {
        countBytes++;
        std::vector<int> HammingCodedByte;
        HammingCodedByte.reserve(20);
        HammingCodedByte = HammingEncoder(byte);
        //write encoded hamming code to file as integers
        int HammingCodeInteger = ConvertBinaryVectorToInt(HammingCodedByte);
        CorrectHammingMessageFile << HammingCodeInteger << "\n";
        std::vector<int> HammingCodeWithError;
        HammingCodeWithError.reserve(20);

        HammingCodeWithError = generateRandomError(HammingCodedByte);
        //print hamming code with error to error file
        HammingCodeInteger = ConvertBinaryVectorToInt(HammingCodeWithError);
        ErrorHammingMessageFile << HammingCodeInteger << "\n";
        char errorByte = HammingDecoder(HammingCodeWithError);
        ErrorMessageFile.put(errorByte);

    }

    // Close the file explicitly
    file.close();
    CorrectHammingMessageFile.close();
    ErrorMessageFile.close();
    ErrorHammingMessageFile.seekg(0, std::ios::beg);// the message that the receiver
receives
    int choice;
    std::cout << "to read hamming code from a file and correct it and write it to
another file press 1\n";
    std::cin >> choice;
    int HammingValue;
    if(choice == 1){
        while(ErrorHammingMessageFile >> HammingValue){
            std::vector<int> CorrectedHammingCode;
            CorrectedHammingCode.reserve(20);
            CorrectedHammingCode = HammingCodeCorrector(HammingValue);
            HammingValue = ConvertBinaryVectorToInt(CorrectedHammingCode);
            CorrectedHammingFile << HammingValue << "\n";
            char CorrectedChar = HammingDecoder(CorrectedHammingCode);
            CorrectedMessageFile.put(CorrectedChar);
        }
    }
    ErrorHammingMessageFile.close();
    CorrectedMessageFile.close();
    std::cout << "the percentage of error is : " <<
((double)count/(countBytes*12))*100 << "%" << std::endl;
    return 0;
}
bool isNotPowerOf2(int a){

    return (a & (a - 1)) != 0;
}

std::vector<int> HammingCodeCorrector(int value) {
    std::vector<int> CorrectedHammingCode;
    CorrectedHammingCode.reserve(20);
    int shmt = -1;
```

```
    for (int i = 0; i <= 12; ++i) {

        shmt++;
        //int shift = value >> shmt;
        int bit = (value >> shmt) & 1;
        CorrectedHammingCode.push_back(bit);
    }

    int XORValue = 0;
    int i = 0;
    shmt = 0;
    int index = 0;
    while((1 << i) <= CorrectedHammingCode.size()){//loop until index is larger than
the size of the vector
        for (int j = 0; j < CorrectedHammingCode.size(); ++j) {

            if((j >> shmt) & 1){//if the index is not a power of 2 and has a 1 at
place n:
                XORValue = XORValue ^ CorrectedHammingCode[j];

            }

        }
        index += XORValue * (1 << i);
        XORValue = 0;
        shmt++;
        i++;

    }
    if(index != 0){
        CorrectedHammingCode[index] = ! CorrectedHammingCode[index];
    }
    return CorrectedHammingCode;
}


int ConvertBinaryVectorToInt(std::vector<int> HammingCode) {
    int HammingInteger = 0;
    for (int j = 0; j < HammingCode.size(); ++j) {
        HammingInteger += HammingCode[j] * (1 << j);
    }

    return HammingInteger;
}

char HammingDecoder(std::vector<int> HammingCode) {
    int RecoveredByte;
    //the bits holding the info are those that are not a multiple of powers of 2
    int ParityIndex = 1;
    int i = 0;
    HammingCode[0] = 9;
    while (ParityIndex <= HammingCode.size()){
        HammingCode[ParityIndex] = 9;
        ParityIndex = (1 << (++i));
    }
    HammingCode.erase(std::remove(HammingCode.begin(), HammingCode.end(), 9),
HammingCode.end());
    for (int j = 0; j < HammingCode.size(); ++j) {
        RecoveredByte += HammingCode[j] * (1 << j);
    }


    return (char)RecoveredByte;
```

```
}

std::vector<int> generateRandomError(std::vector<int> HammingCode) {
    count++;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(1, 12);

    int randomValue = distr(gen);
    HammingCode[randomValue] = !HammingCode[randomValue];
    return HammingCode;
}

std::vector<int> HammingEncoder(char byte) {

    std::vector<int>HammingCode;
    HammingCode.reserve(20);
    int shmt = -1;
    for (int i = 0; i <= 12; ++i) {
        if(isNotPowerOf2(i)){
            shmt++;
            int bit = (byte >> shmt) & 1;
            HammingCode.push_back(bit);
        }
        else{
            HammingCode.push_back(0);
        }
    }
    //now to fill the bits in the 2^n indexes
    //the steps are:
    //1. shift the vector indexes by a shmt := 0
    //  a. test for ___1
    //  b. test for __1_
    //  c. test for _1__
    //  d. test for 1___
    //  where shmt(shift amount) is changed each iteration for the 4 cases
    //2. mask by 1
    //3. xor it with the previous bit
    //4. update the parity indexes
    int i = 0;
    shmt = 0;
    int XORValue = 0;
    while((1 << i) <= HammingCode.size()){//loop until index is larger than the size
of the vector
        int index = 1 << i;// to fill the indexes with multiple of 2
        for (int j = 0; j < HammingCode.size(); ++j) {
            if(isNotPowerOf2(j)){
                if((j >> shmt) & 1){//if the index is not a power of 2 and has a 1 at
place n:
                    XORValue = XORValue ^ HammingCode[j];

                }
            }
        }
        HammingCode[index] = XORValue;
        XORValue = 0;
        shmt++;
        i++;
    }

    return HammingCode;
}
```