COM 5140 Error–Correcting Codes

Project 2

111062109 黃庭曜

Table of Contents

§1 Description

§1.1 Construction of GF(64)

To make the decoding process smooth. The class “GF64” is declared, and the operators are overloaded.

For operator “+”, the calculation is the same as doing an XOR operation on the binary values. For example, $11(\alpha^3+\alpha+1)+6(\alpha^2+\alpha)=13(\alpha^3+\alpha^2+1)$. Operator “−“ is the same as “+”.

For operator “×”, the most efficient way is to use the power table and the log table. The power table stores the values α^i for all i, and the log table stores the values i for all α^i. For example, `pow_table[4] = 16, log_table[16] = 4`. Then, if we want to calculate $x \times y$, we can transfer x to α^i, y to α^j, then $x \times y = \alpha^i \times \alpha^j = \alpha^{(i+j) \mod 63}$. Then find the original value for $\alpha^{(i+j) \mod 63}$ via the power table.

For operator “/”, similar to the operator “*”, changes to $x/y = \alpha^i/\alpha^j = \alpha^{(i-j) \mod 63}$.

§1.2 Construction of GF64 Polynomial

Since we have lots of polynomials to perform operations, The class “GF64_poly” is declared, and the operators are overloaded.

Vectors make the basic storage for the coefficients. We can access the coefficient by inserting the index i, which is designed to be the same as the corresponding x^i’s coefficient.

For operator “+”, the initial output polynomial degree is the maximum among the addend and the augend. Then, add the two polynomials; the addition for the GF64 is defined in the GF64 class. Operator “−“ is the same as “+”. Note that after the calculation, removing the padding 0s is essential to preserve the correct degree.

For operator “×” (two polynomials), the output polynomial degree is the sum of the multiplicand and the multiplier. The result can be calculated using the convolution.

For operator “×” (with a GF64 number), the output polynomial degree is the same as the multiplicand. The result can be calculated by multiplying the number sequentially with the multiplicand.

For operator “/” (two polynomials), we need to check if the divisor is 0. If the divisor is 0, return an error. If the divisor has a degree less than that of the dividend, the result is 0. Then, we perform the division by dividing the highest term of the dividend by the divisor. Sequentially perform by saving the quotient, calculating the remainder, and dividing the remainder by the divisor. The process stops when the remainder has a smaller degree than the divisor.

For operator (), we insert a GF64 value to calculate the polynomial result. We only need to sequentially calculate the multiplication result.

For the function "differentiate", we have to perform a formal derivative on the polynomial. We have to be careful that $\frac{d}{dx} p_l x^l = l \times p_l x^{l-1}$, l is defined as adding 1 l times. In GF64, $1 + 1 = 0$, we can simplify to $l = \begin{cases} 1, \ l \text{ is odd} \\ 0, \ l \text{ is even} \end{cases}$, so we only need to keep the odd terms of the original polynomial and shift the terms.

The last function is "mod_x21", which means perform mod x^{21} on the polynomial. We only have to drop all the terms with a degree greater than 20 and remove the padding 0s.

§1.3 Decoder

The decode process is:

1. Calculate the erasure locator

For $I_0 = \{i \,|\, R_i = * \}$, the erasure locator polynomial is defined as $\sigma_0(x) = \prod_{i \in I_0} (1 - \alpha^i x)$. Since in GF64, addition is equal to subtraction, $\sigma_0(x) = \prod_{i \in I_0} (1 + \alpha^i x)$.

2. Construct the modified received vector

We have to initialize the erasure symbols to 0, i.e., construct the modified received vector:

$$R_i' = \begin{cases} R_i, & i \notin I_0 \\ 0, & i \in I_0 \end{cases}$$

3. Calculate syndromes:

The syndrome is defined as $S_j = \sum_{i=0}^{n-1} R_i' \alpha^{ij}$, and the syndrome polynomial is defined as:

$$S(x) = S_1 + S_2 x + \ldots + S_r x^{r-1}$$

The calculation is simple under the defined GF64 class.

4. Perform the Euclidean algorithm

The error set is defined as $I_1 = \{i \,|\, R_i \neq * \wedge R_i \neq C_i\}$, and define the set $I = I_0 \cup I_1$. We have to calculate the error locator polynomial $\sigma_1(x) = \prod_{i \in I_1} (1 - \alpha^i x) = \prod_{i \in I_1} (1 + \alpha^i x)$. The error–and–erasure locator polynomial is defined as $\sigma(x) = \sigma_0(x)\sigma_1(x)$. The key equation is:

$$\sigma(x) S(x) \equiv \omega(x) \mod x^r$$

Since $\sigma_0(x)$ is known, the modified syndrome polynomial is defined as

$$S_0(x) = \sigma_0(x) S(x) \mod x^r$$

The key equation to be solved now becomes:

$$\sigma_1(x)S_0(x) \equiv \omega(x) \mod x^r$$

where $\omega(x)$ is the error-and-erasure evaluator polynomial.

Since $\deg(\sigma_1(x)) = e_1 \le \lfloor \frac{r-e_0}{2} \rfloor$ and $\deg(\omega(x)) \le e_0 + e_1 - 1 \le \lceil \frac{r+e_0}{2} \rceil - 1$, define:

$$\mu = \lfloor \frac{r-e_0}{2} \rfloor \text{ and } \nu = \lceil \frac{r+e_0}{2} \rceil - 1 \Rightarrow \mu + \nu = r - 1 = \deg(x^r) - 1$$

In C++, integer division will take its floor as the result. We can rewrite $\nu = \lfloor \frac{r+e_0+1}{2} \rfloor - 1$ since r & e_0 are integers.

Apply extended Euclid's algorithm to x^r and $S_0(x)$. Stop when $\deg(r_j(x)) \le \mu$ and $\deg(v_j(x)) \le \mu$. Then obtain $\sigma_1(x) = v_j(x)$ and $w(x) = r_j(x)$. We then have $\sigma(x) = \sigma_0(x)\sigma_1(x)$.

The completion of the algorithm requires operations such as $S_0(x) \mod x^{21}$, polynomial addition (subtraction), multiplication, division, and remainder calculation (can be implemented with a division and a subtraction operation). These methods are provided in the class GF64_poly.

5. Calculate the error vector with the error locator and the error evaluator.

The error vector is defined as $\mathbb{E} = R' - C$. Define $\sigma(x) = \hat{\sigma}_0 + \hat{\sigma}_1 x + \ldots + \hat{\sigma}_d x^d$ By the Time-Domain Completion, we first check if $\hat{\sigma}_0 = \sigma(0)$ is zero. If it is, the received vector is not decodable. Then, if $\deg(\omega(x)) \ge e_0 + \deg(\sigma_1(x))$, the received vector is also not decodable. Then, we go through all the error positions to calculate the error vector.

$$E_i = -\omega(\alpha^{-i})/\sigma'(\alpha^i) = \omega(\alpha^{-i})/\sigma'(\alpha^i) \text{ if } \sigma(\alpha^{-i}) = 0 \wedge \sigma'(\alpha^{-i}) \neq 0, \text{ else } 0$$

Count the number of non-zero error values to check if it matches the degree of $\sigma(x)$. If matched, the error vector is our desired result. If not, decode fails. Note that $\sigma'(x)$ is the formal derivative of the polynomial $\sigma(x)$, which is defined in the class GF64_poly.

6. Calculate the correct codeword.

Since $\mathbb{E} = R' - C$, Calculate $C = R' - \mathbb{E} = R' + \mathbb{E}$.

§2 Discussion

§2.1 Decode only with errors

With the number of errors less than or equal to 10, the decoder is able to correct to the original codeword. However, with the number of errors more than 10, the decoder will mostly give up, or with a small probability, a different codeword has a Hamming distance with the corrupted codeword less than or equal to 10, the decoder will decode to it.

§2.2 Decode only with erausres

With the number of erasures less than or equal to 21, the decoder is able to correct to the original codeword. However, with the number of erasures more than 21, the decoder should give up, or the process will lead to an incorrect result. (even not a codeword)

§2.3 Decode with error and erausres

The decoder has the ability to recover any codeword with $e_0 + 2 \times e_1 \leq 21$, where e_0 is the number of erasures and e_1 is the number of errors. For $e_0 + 2 \times e_1 > 21$, the decoder may output a codeword that has an extended Hamming distance with the corrupted codeword less than or equal to 10.5, or simply give up.

§2.4 Testing

To test the functionality of the decoder, three extra files are constructed.

`encoder.cpp`: The file creates random codewords. Codewords can be made by a randomly generated message (in polynomial form) times the generator polynomial. Since we only need to check the functionality of the decoder, we don't care if the encoder is systematic.

`error_maker.cpp`: The file creates random errors and erasures with the input codeword. We can specify the number of errors and erasures, and the code will output the corrupted codeword.

`calculate_distance.cpp`: The file inputs two words and calculates the difference and the erasures between them. This helps to verify that the decoded codeword is the same as the original codeword, and the result has the same distance as the designed number of errors and erasures.

The codes are attached in the appendix. (The classes GF64 & GF64_poly are identical to the ones in our decoder, so that they will be omitted in the appendix.)

§3 Code with comments

```
#include <iostream>
#include <cstdio>
#include <vector>

// Power table for GF(64)
int pow_table[63] = {1, 2, 4, 8, 16, 32, 3, 6, 12, 24, 48, 35,
                     5, 10, 20, 40, 19, 38, 15, 30, 60, 59, 53,
                     41, 17, 34, 7, 14, 28, 56, 51, 37, 9, 18,
                     36, 11, 22, 44, 27, 54, 47, 29, 58, 55, 45,
                     25, 50, 39, 13, 26, 52, 43, 21, 42, 23, 46,
                     31, 62, 63, 61, 57, 49, 33};
// Logarithm table for GF(64)
int log_table[64];
// Coefficients of the generator polynomial for the Reed-Solomon code
const int gen_poly[22] = {58, 62, 59, 7, 35, 58, 63, 47, 51, 6, 33,
                            43, 44, 27, 7, 53, 39, 62, 52, 41, 44, 1};
class GF64 {
    private:
        int value;
    public:
        GF64() { this->value = 0; }
        GF64(int value) { this->value = value; }
        // Add the polynomial
        GF64 operator+(const GF64& other) const {
            // Simple XOR operation
            return GF64(value ^ other.value);
        }
        // Multiply the polynomial
        GF64 operator*(const GF64& other) const {
            // If one of the two numbers is 0, the result is 0
            if(value == 0 || other.get_value() == 0) return GF64(0);
            // Use the log table to times two GF64 numbers
            int log_value = log_table[value] + log_table[other.get_value()];
            return GF64(pow_table[log_value % 63]);
        }
        // Divide the polynomial
        GF64 operator/(const GF64& other) const {
            // If the other number is 0, the result is undefined
            if(other.get_value() == 0)
                throw std::invalid_argument("Division by zero");
            // If the number is 0, the result is 0
            if(value == 0) return GF64(0);
            // Use the log table to divide two GF64 numbers
            int log_value = log_table[value] - log_table[other.get_value()] + 63;
            return GF64(pow_table[log_value % 63]);
        }
        // Assign the value of the GF64 number
        GF64 operator=(const GF64& other) {
            this->value = other.value;
            return *this;
        }
        // Get the value of the GF64 number
        int get_value() const {
            return value;
        }
};
```

```
class GF64_poly {
    private:
       // The coefficients of the polynomial (The first element is the constant term)
        std::vector<GF64> coefficients;
        int degree;
    public:
        GF64_poly() {
            degree = 0;
            coefficients.resize(1);
            coefficients[0] = GF64(0);
        }
        GF64_poly(const std::vector<GF64>& coefficients) {
            this->coefficients = coefficients;
            this->degree = coefficients.size() - 1;
            while(degree > 0 && coefficients[degree].get_value() == 0) degree--;
        }
        // Set the coefficients of the polynomial
        GF64_poly set_coefficients(int index, GF64 value) {
            // If the index is greater than the degree, resize the polynomial
            if(index > degree){
                degree = index;
                coefficients.resize(degree + 1);
            }
            coefficients[index] = value;
            return *this;
        }
        // Get the degree of the polynomial
        int get_degree() const {
            return degree;
        }
        // Add two polynomials
        GF64_poly operator+(const GF64_poly& other) const {
            // The result's degree is the maximum degree of the two polynomials
            std::vector<GF64> result(std::max(degree, other.degree) + 1, GF64(0));
            for(int i = 0; i <= degree; i++) result[i] = coefficients[i];
            for(int i = 0; i <= other.degree; i++) result[i] = result[i] +
other.coefficients[i];
            // Remove leading zeros while keeping at least one term
            while(result.size() > 1 && result.back().get_value() == 0)
                 result.pop_back();
            return GF64_poly(result);
        }
        // Multiply a polynomial and a GF64 number
        GF64_poly operator*(const GF64& other) const {
            // The result's degree is the degree of the polynomial
            std::vector<GF64> result(degree + 1);
            for(int i = 0; i <= degree; i++) {
                result[i] = coefficients[i] * other;
            }
            // Remove leading zeros while keeping at least one term
            while(result.size() > 1 && result.back().get_value() == 0) {
                result.pop_back();
            }
            return GF64_poly(result);
        }
        // Multiply two polynomials
        GF64_poly operator*(const GF64_poly& other) const {
            // Initialize result with zeros, the result's degree is the sum of the 
degrees of the two polynomials
            std::vector<GF64> result(degree + other.degree + 1, GF64(0));
            // Perform polynomial multiplication
            for(int i = 0; i <= degree; i++) {
                if(coefficients[i].get_value() != 0) {  // Skip zero terms
                    for(int j = 0; j <= other.degree; j++) {
                        if(other.coefficients[j].get_value() != 0) {  // Skip zero 
terms
                            result[i+j] = result[i+j] + coefficients[i] * 
other.coefficients[j];
                        }
                    }
                }
            }
            // Remove leading zeros while keeping at least one term
            while(result.size() > 1 && result.back().get_value() == 0) 
result.pop_back();
            return GF64_poly(result);
        }
        // Divide a polynomial by another polynomial
        GF64_poly operator/(const GF64_poly& other) const {
            // Check for division by zero polynomial
            if(other.degree == 0 && other.coefficients[0].get_value() == 0) {
                std::cout << "Error: Division by zero polynomial" << std::endl;
                exit(1);
            }
            // If the degree of the polynomial is less than the degree of the other 
polynomial, the result is 0
            if(degree < other.degree) {
                return GF64_poly();
            }
            // Initialize the quotient with zeros
            // The result's degree is the degree of the polynomial minus the degree 
of the other polynomial
            std::vector<GF64> quotient(degree - other.degree + 1, GF64(0));
            std::vector<GF64> remainder = coefficients;
            // Perform polynomial division
            for(int i = degree; i >= other.degree; i--) {
                // If the leading coefficient of the remainder is not 0
                if(remainder[i].get_value() != 0) {
                    GF64 coef = remainder[i] / other.coefficients[other.degree];
                    quotient[i - other.degree] = coef;
                    // Update the remainder
                    // Y = X * Q + R
                    for(int j = 0; j <= other.degree; j++) {
                        remainder[i - j] = remainder[i - j] + 
                            other.coefficients[other.degree - j] * coef;
                    }
                }
            }
            // Remove leading zeros while keeping at least one term
            while(quotient.size() > 1 && quotient.back().get_value() == 0) {
                quotient.pop_back();
            }
            return GF64_poly(quotient);
        }
        GF64_poly operator=(const GF64_poly& other) {
            this->coefficients = other.coefficients;
            this->degree = other.degree;
            return *this;
        }
        GF64_poly operator=(const std::vector<GF64>& coefficients) {
            this->coefficients = coefficients;
            this->degree = coefficients.size() - 1;
            // Remove leading zeros while keeping at least one term
            while(degree > 0 && coefficients[degree].get_value() == 0) {
                degree--;
            }
            return *this;
        }
        // Evaluate the polynomial at a given number
        GF64 operator()(const GF64& x) const {
            // If the number is 0, the result is the constant term
            if(x.get_value() == 0) return coefficients[0];
            GF64 result(0), power(1);
            for(int i = 0; i <= degree; i++){
                result = result + coefficients[i] * power;
                // Update the power (a^i)
                power = power * x;
            }
            return result;
        }
        GF64_poly differentiate() const {
            GF64_poly result;
            // Initialize the result with the first coefficient
            result.set_coefficients(0, coefficients[1]);
            // If the degree is even, the result is the coefficient of the polynomial
            // Otherwise, the result is 0 (1+1=0 in GF(64))
            for(int i = 1; i < degree; i++){
                if(i % 2 == 0) result.set_coefficients(i, coefficients[i+1]);
                else result.set_coefficients(i, 0);
            }
            return result;
        }
        void print() const {
            for(int i = 0; i < coefficients.size(); i++)
                printf("%d ", coefficients[i].get_value());
            printf("\n");
        }
        bool is_zero() const {
            return degree == 0 && coefficients[0].get_value() == 0;
        }
        GF64_poly mod_x21() const {
            // The result is simply the polynomial dropping all the terms with degree
greater than 20
            GF64_poly result;
            for(int i = 20; i >= 0; i--){
                if(coefficients[i].get_value() != 0)
                    result.set_coefficients(i, coefficients[i]);
            }
            return result;
        }
};
```

```
class ReedSolomonDecoder {
    private:
        static const int n = 63;  // Code length
        static const int k = 42;  // Message length
        static const int t = 10;  // Error correction capability

    // Calculate syndromes including erasure information
    GF64_poly calculateSyndromes(const std::vector<GF64>& received,
                                 const std::vector<bool>& erasures) {
        std::vector<GF64> syndromes;
        syndromes.resize(21);
        for (int j = 0; j <= 20; j++) {
            // Syndrome S_j = sum(a^ij * c_i), j = 1~21
            GF64 syndrome(0);
            for (int i = 0; i < n; i++) {
                if (received[i].get_value() != 0) {
                    GF64 alpha(pow_table[(i * (j+1)) % 63]);
                    syndrome = syndrome + received[i] * alpha;
                }
            }
            // Save the syndrome with shifted ( syndromes[j] = s_(j+1) )
            syndromes[j] = syndrome;
        }
        // Return the syndrome polynomial
        return GF64_poly(syndromes);
    }

    // Calculate erasure locator polynomial
    GF64_poly calculateErasureLocator(const std::vector<bool>& erasures) {
        // Initialize the erasure locator polynomial
        GF64_poly erasureLocator(std::vector<GF64>{GF64(1)});
        for (int i = 0; i < n; i++) {
            if (erasures[i]) {
                // Multiply by (1 + a^i * x)
                std::vector<GF64> factor = {GF64(1), GF64(pow_table[i])};
                GF64_poly factorPoly(factor);
                erasureLocator = erasureLocator * factorPoly;
            }
        }
        // If the degree of the erasure locator polynomial is greater than 21, the
decoding fails
        if(erasureLocator.get_degree() > 21){
            std::cout << "Error: Erasure locator polynomial degree exceeds 21" <<
std::endl;
            exit(1);
        }
        return erasureLocator;
    }
    // Euclidean algorithm
    std::pair<GF64_poly, GF64_poly> euclideanAlgorithm(
        const GF64_poly& syndromes,
        const GF64_poly& erasureLocator) {

        // Initialize polynomials
        GF64_poly S_0(erasureLocator * syndromes); // Modified Syndrome Polynomial
        // mod x^21
        GF64_poly x21(std::vector<GF64>{GF64(0), GF64(0), GF64(0), GF64(0),
        GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0),
        GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0), GF64(0),
        GF64(0), GF64(1)});

        S_0 = S_0.mod_x21();
        int num_of_erasures = erasureLocator.get_degree();
        // mu = lower bound of (r-e_0)/2
        int mu = (21 - num_of_erasures) / 2;
        // nu = upper bound of (r+e_0)/2 - 1
        int nu = (21 + num_of_erasures + 1) / 2 - 1;
        // Initialize the polynomials
        std::vector<GF64_poly> R, Q, U, V;
        // R_0 = x^r = x^21
        R.push_back(x21);
        R.push_back(S_0);
        // Q_0 = 0 (Don't care)
        Q.push_back(GF64_poly(std::vector<GF64>{0}));
        // Q_1 = 0 (Don't care)
        Q.push_back(GF64_poly(std::vector<GF64>{0}));
        // S_0 = 1
        U.push_back(GF64_poly(std::vector<GF64>{1}));
        // S_1 = 0
        U.push_back(GF64_poly(std::vector<GF64>{0}));
        // T_0 = 0
        V.push_back(GF64_poly(std::vector<GF64>{0}));
        // T_1 = 1
        V.push_back(GF64_poly(std::vector<GF64>{1}));
        while(R[R.size() - 1].get_degree() > nu || V[V.size() - 1].get_degree() > mu)
        {
            // Q_i = R_(i-2) / R_(i-1)
            GF64_poly Q_i = R[R.size() - 2] / R[R.size() - 1];
            Q.push_back(Q_i);
            // R_i = R_(i-2) + R_(i-1) * Q_i
            R.push_back(R[R.size() - 2] + R[R.size() - 1] * Q_i);
            // U_i = U_(i-2) + U_(i-1) * Q_i
            U.push_back(U[U.size() - 2] + U[U.size() - 1] * Q_i);
            // V_i = V_(i-2) + V_(i-1) * Q_i
            V.push_back(V[V.size() - 2] + V[V.size() - 1] * Q_i);
        }
        // Return the error locator and error evaluator
        return std::make_pair(V[V.size() - 1], R[R.size() - 1]);
    }
    // Error correction
    std::pair<bool, GF64_poly> correctErrors(
        GF64_poly& erasureLocator, GF64_poly& errorLocator,
        GF64_poly& error_and_erasure_Evaluator
    )
    {
        // Initialize the error locator polynomial
        GF64_poly error_and_erasures_Locator = errorLocator * erasureLocator;
        bool is_correctable = false;
        std::vector<GF64> err(n);
        // Time domain completion
        // If the error locator polynomial is 0, the decoding fails
        if(error_and_erasures_Locator(0).get_value() == 0)
            return std::make_pair(false, err);
        // deg(w) < e_0 + deg(erasureLocator)
        if(error_and_erasure_Evaluator.get_degree() >= errorLocator.get_degree() + 
erasureLocator.get_degree()) return std::make_pair(false, err);
        int count = 0;
        // Get the formal derivative of the error locator polynomial
        GF64_poly error_and_erasures_Locator_derivative = 
error_and_erasures_Locator.differentiate();
        // Error value
        for(int i = 0; i < n; i++){
            GF64 alpha = pow_table[(63 - i) % 63];
            if(error_and_erasures_Locator(alpha).get_value() == 0 && 
error_and_erasures_Locator_derivative(alpha).get_value() != 0){
                count++;
                err[i] = error_and_erasure_Evaluator(alpha) / 
error_and_erasures_Locator_derivative(alpha);
            }
            else err[i] = GF64(0);
        }
        // If the number of error is equal to the degree of the error locator 
polynomial, the error is correctable
        is_correctable = (count == error_and_erasures_Locator.get_degree());
        return std::make_pair(is_correctable, GF64_poly(err));
    }
public:
    std::pair<bool, GF64_poly> decode(const std::vector<GF64>& received,
                        const std::vector<bool>& erasures = std::vector<bool>()) {
        GF64_poly received_poly(received);
        // Calculate syndromes
        GF64_poly syndromes = calculateSyndromes(received, erasures);
        if(syndromes.is_zero()) return std::make_pair(true, received_poly);
        // Calculate erasure locator polynomial
        GF64_poly erasureLocator = calculateErasureLocator(erasures);
        // Apply the Euclidean algorithm, return error locator and error evaluator
        std::pair<GF64_poly, GF64_poly> result = euclideanAlgorithm(syndromes, 
erasureLocator);
        // Error correction
        std::pair<bool, GF64_poly> error_correction_result = 
correctErrors(erasureLocator, result.first, result.second);
        // codeword = received + error
        GF64_poly codeword = received_poly + error_correction_result.second;
        // Return the result
        return std::make_pair(error_correction_result.first, codeword);
    }
};
```

```
int main() {
    log_table[0] = 0;
    for(int i = 1; i < 64; i++) {
        log_table[pow_table[i-1]] = i-1;
    }
    std::vector<GF64> received(63);
    std::vector<bool> erasures(63);
    for(int i = 0; i < 63; i++) {
        char c; int x;
        scanf(" %c", &c);
        if(c == '*'){
            // If the received codeword is an erasure, set the erasure to true
            received[i] = GF64(0);
            erasures[i] = true;
        }
        else{
            // If the received codeword is not an erasure, set the erasure to false
            ungetc(c, stdin);
            scanf("%d", &x);
            received[i] = GF64(x);
            erasures[i] = false;
        }
    }

    ReedSolomonDecoder decoder;
    // Decode the received codeword
    std::pair<bool, GF64_poly> decoded = decoder.decode(received, erasures);
    if(decoded.first){
        // Print the decoded codeword
        decoded.second.print();
    }
    else{
        // If the decoding fails, print "give up"
        printf("give up\n");
    }

    return 0;
}
```

§A Appendix

§A.1 encoder.cpp

```
class ReedSolomonEncoder {
private:
    static const int n = 63;  // Code length
    static const int k = 42;  // Message length
    static const int t = 10;  // Error correction capability

public:
    // Create generator polynomial
    GF64_poly createGeneratorPolynomial() {
        std::vector<GF64> gen_coeffs(22);
        for(int i = 0; i < 22; i++) {
            gen_coeffs[i] = GF64(gen_poly[i]);
        }
        return GF64_poly(gen_coeffs);
    }
    // Encode a message into a codeword
    std::vector<GF64> encode(const std::vector<GF64>& message) {
        if (message.size() != k) {
            throw std::invalid_argument("Message length must be " +
std::to_string(k));
        }

        // Create message polynomial
        GF64_poly message_poly(message);

        // Create generator polynomial
        GF64_poly gen_poly = createGeneratorPolynomial();

        // Multiply polynomials
        GF64_poly codeword_poly = message_poly * gen_poly;

        // Convert to vector and ensure length is n
        std::vector<GF64> codeword = codeword_poly.coefficients;
        codeword.resize(n, GF64(0));  // Pad with zeros if necessary

        return codeword;
    }

    // Encode a message from raw integers
    std::vector<GF64> encodeFromInts(const std::vector<int>& message) {
        if (message.size() != k) {
            throw std::invalid_argument("Message length must be " +
std::to_string(k));
        }
        std::vector<GF64> gfMessage;
        for (int value : message) {
            gfMessage.push_back(GF64(value));
        }

        return encode(gfMessage);
    }
};
```

```
int main() {
    // Initialize logarithm table
    srand(time(0));
    log_table[0] = 0;
    for(int i = 1; i < 64; i++) log_table[pow_table[i-1]] = i-1;
    // Create encoder
    ReedSolomonEncoder encoder;
    // Randomly generate 42 symbols
    std::vector<int> message(42);
    for(int i = 0; i < 42; i++) message[i] = rand() % 64;
    // Encode the message
    std::vector<GF64> codeword = encoder.encodeFromInts(message);
    // Output the complete codeword
    for (const auto& symbol : codeword) std::cout << symbol.get_value() << " ";
    std::cout << std::endl;
    return 0;
}
```

§A.2 error_maker.cpp

```
void initialize_tables() {
    log_table[0] = 0;
    for(int i = 1; i < 64; i++) log_table[pow_table[i-1]] = i-1;
}
std::vector<GF64> generate_corrupted_codeword(const std::vector<GF64>& original,
                                            int num_errors, int num_erasures) {
    std::vector<GF64> corrupted = original;
    std::vector<bool> used_positions(63, false);
    // Random number generator
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 62);
    // Generate erasures
    for(int i = 0; i < num_erasures; i++) {
        int pos;
        do {
            pos = dis(gen);
        } while(used_positions[pos]);
        used_positions[pos] = true;
        corrupted[pos] = GF64(-1);  // Use -1 to mark erasures
    }
    // Generate errors
    for(int i = 0; i < num_errors; i++) {
        int pos;
        do {
            pos = dis(gen);
        } while(used_positions[pos]);
        used_positions[pos] = true;
        // Generate random non-zero error value
        int error_value;
        do {
            error_value = dis(gen) + 1;  // +1 to avoid 0
        } while(error_value > 63);
        corrupted[pos] = corrupted[pos] + GF64(error_value);
    }
    return corrupted;
}
```

```
int main() {
    initialize_tables();
    // Read original codeword
    std::vector<GF64> original(63);
    std::cout << "Enter the original codeword (63 values, 0-63):\n";
    for(int i = 0; i < 63; i++) {
        int x;
        std::cin >> x;
        if(x < 0 || x > 63) {
            std::cout << "Invalid input: values must be between 0 and 63\n";
            return 1;
        }
        original[i] = GF64(x);
    }
    // Read number of errors and erasures
    int num_errors, num_erasures;
    std::cout << "Enter number of errors: ";
    std::cin >> num_errors;
    std::cout << "Enter number of erasures: ";
    std::cin >> num_erasures;
    // Generate corrupted codeword
    std::vector<GF64> corrupted = generate_corrupted_codeword(original, num_errors,
num_erasures);
    // Output the corrupted codeword
    std::cout << "Corrupted codeword:\n";
    for(int i = 0; i < 63; i++) {
        if(corrupted[i].get_value() == -1) {
            std::cout << "* ";  // Print * for erasures
        }
        else std::cout << corrupted[i].get_value() << " ";
    }
    std::cout << "\n";
    return 0;
}
```

§A.3 calculate_distance.cpp

```
// Calculate distance between two codewords
// Returns a pair of (total_distance, num_errors, num_erasures)
std::pair<int, std::pair<int, int>> calculate_distance(const std::vector<GF64>&
word1, const std::vector<GF64>& word2) {
    int total_distance = 0;
    int num_errors = 0;
    int num_erasures = 0;
    for(int i = 0; i < 63; i++) {
        if(word1[i].get_value() == -1 || word2[i].get_value() == -1) {
            // If either position is an erasure
            total_distance += 1; num_erasures++;
        }
        else if(word1[i].get_value() != word2[i].get_value()) {
            // If values are different (error)
            total_distance += 2; num_errors++;
        }
    }
    return {total_distance, {num_errors, num_erasures}};
}
```

```cpp
int main() {
    // Read first codeword
    std::vector<GF64> word1(63);
    std::cout << "Enter first codeword (63 values, use * for erasures):\n";
    for(int i = 0; i < 63; i++) {
        char c;
        std::cin >> c;
        if(c == '*') word1[i] = GF64(-1);  // -1 represents erasure
        else {
            std::cin.unget();
            int x;
            std::cin >> x;
            if(x < 0 || x > 63) {
                std::cout << "Invalid input: values must be between 0 and 63\n";
                return 1;
            }
            word1[i] = GF64(x);
        }
    }
    // Read second codeword
    std::vector<GF64> word2(63);
    std::cout << "Enter second codeword (63 values, use * for erasures):\n";
    for(int i = 0; i < 63; i++) {
        char c;
        std::cin >> c;
        if(c == '*') word2[i] = GF64(-1);  // -1 represents erasure
        else {
            std::cin.unget();
            int x;
            std::cin >> x;
            if(x < 0 || x > 63) {
                std::cout << "Invalid input: values must be between 0 and 63\n";
                return 1;
            }
            word2[i] = GF64(x);
        }
    }
    // Calculate and print distance
    auto [total_distance, counts] = calculate_distance(word1, word2);
    auto [num_errors, num_erasures] = counts;

    std::cout << "Distance calculation results:\n";
    std::cout << "Total distance: " << total_distance << "\n";
    std::cout << "Number of errors: " << num_errors << "\n";
    std::cout << "Number of erasures: " << num_erasures << "\n";

    return 0;
}
```