Reed Solomon Code Decoder

109061217 林峻霆

1. System Design

這次要實作的是一個（63, 42）Reed Solomon Code Decoder。為了方便進行測試，我順便連Encoder的部分也一起實作。在實作Encoder和Decoder前，因為（63, 42）Reed Solomon Code Decoder操作在GF(64)上，且要搭配GF(64)上的多項式運算，所以在實作兩者之前，我先實作GF(64)以及多項式(Polynomial)的部分，方便主程式(Encoder和Decoder)的撰寫。

- Encoder

Encoder的部分先用rand()隨機生成information bits I(x)，接著用generator polynomial g(x)進行encoding，C(x) = I(x)g(x)。接著在產生的codeword中隨機加入error會erasure，並輸出到一個額外的檔案(testcase.txt)。

- Decoder

Decoding的步驟如下：

a. 將received vector R(x)中的erasure(*)改為0，並計算erasure locator polynomial $\sigma_0(x)$。若此時的erasure數量e_0已經超過r，則直接declare failure不繼續進行下面的步驟。

b. 將修改後的received vector R'(x)對g(x)取餘數，減少後續compute syndrome時不必要的計算。接著compute syndrome，$S_j = R'(\alpha^j)$。

c. 接著計算modified syndrome polynomial，$S_0(x) = S(x)\sigma_0(x)\ mod(x^r)$

d. 設定Extended Euclidean Algorithm(EEA)的停止條件

$$\mu = \left\lfloor \frac{r - e_0}{2} \right\rfloor \ and \ \nu = r - 1 - \mu$$

接著執行$EEA(x^r, S_0(x), \mu, \nu)$，找到error locator polynomial $\sigma_1(x)$ 與 error-and-erasure evaluator polynomial $\omega(x)$。

e. 執行Time domain approach，找到error polynomial E(x)，最後C(x) = R(x) – E(x)。若在過程中發現error和erasure的數量超過可解範圍，則declare failure。

2. Discussion

經過自己測試以及與助教進行demo後，程式基本上沒有任何問題。當有e0個erasure以及e1個error時，若e0＋2*e1 ≤ r，則必定能將error與erasure修正掉，反之則會declare failure。我認為整個程式仍能修正的部分有兩點：

a. 運用類似於project 1的方式，將傳輸過程模擬於一個erasure channel中，並將Eb / N0對錯誤率作圖，而非直接隨機產生erasure與error。這樣應該比較符合實際狀況。

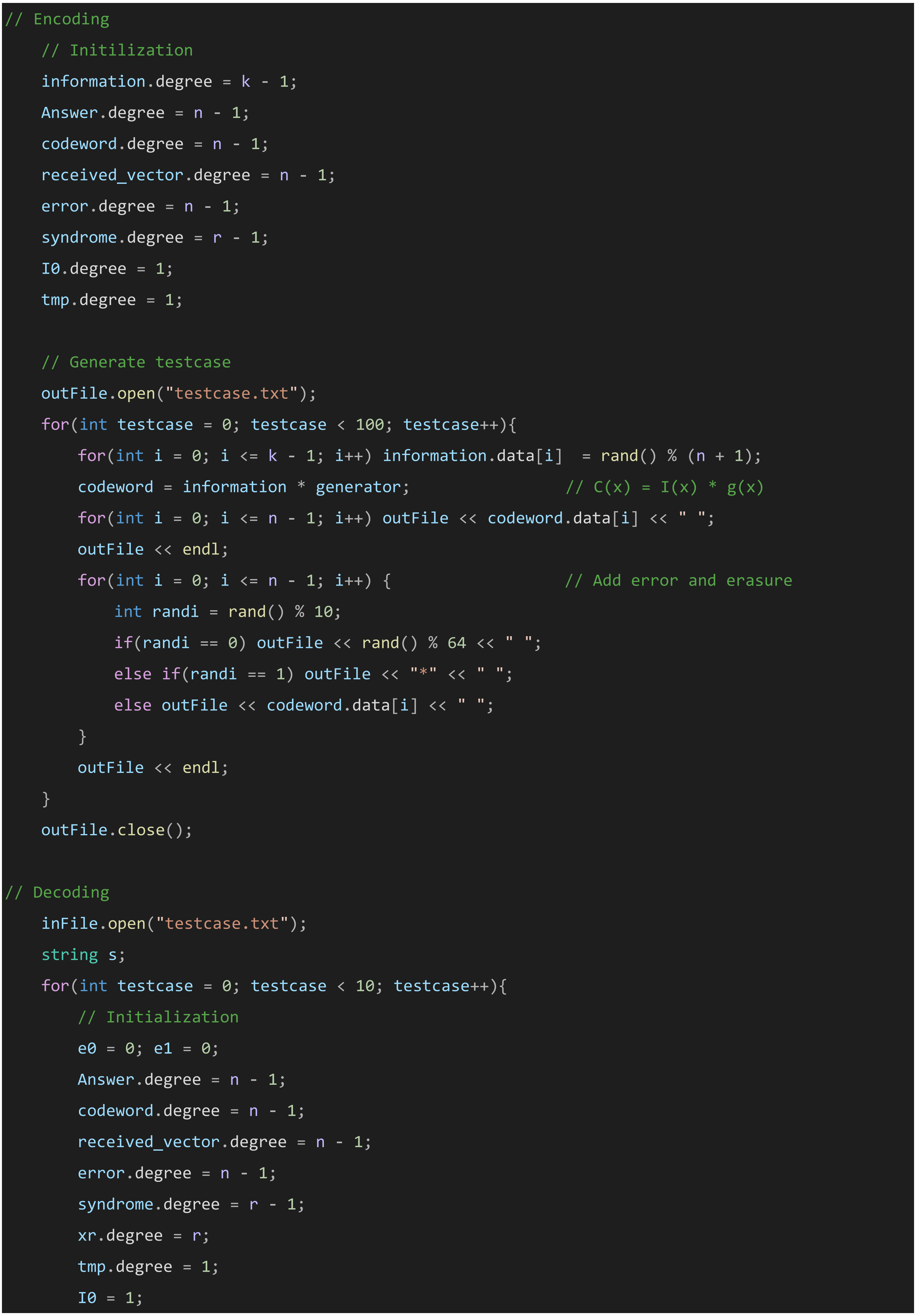
b. Polynomial這個data structure能實作的更general，不應該僅限於GF(64)。Polynomial與實作大數(Big Number)的運算有異曲同工之處，在密碼學的一些程式實作上(RSA, Rabin等)應該也能有所幫助。未來希望能更優化這個class，不論是內部運算的效率或是泛用性等部分。

3. Program

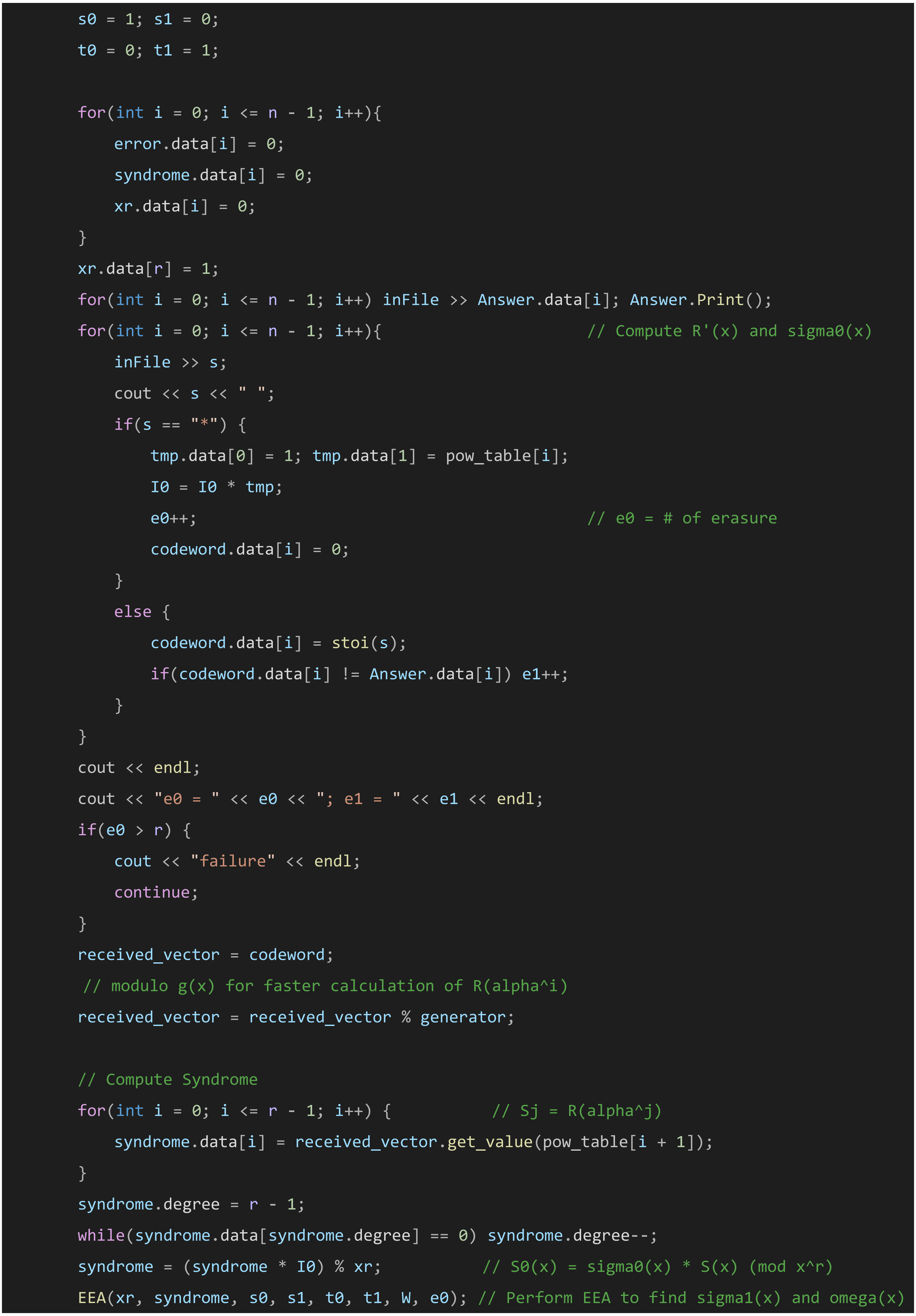
程式主要分為兩部分：main function 以及 polynomial。main function 的部分主要包含 encoding 以及 decoding 兩個步驟。而 polyomial 的部分則是實作 polynomial 這個 data structure 的一些細節(例如：加減乘除、代值、取餘數等)

a. Main function

```
#include <iostream>
#include <stdlib.h>
#include <algorithm>
#include <vector>
#include <string.h>
#include <fstream>
#include <time.h>
#include "Polynomial.h"

using namespace std;

// Extended Euclidean Algorithm
void EEA(Polynomial P, Polynomial Q, Polynomial& s0, Polynomial& s1, Polynomial& t0,
Polynomial& t1, Polynomial& W, int e0);

int main(void){
    int x;
    int e0, e1;
    int count;
    bool failure = false;
    ifstream inFile;
    ofstream outFile;
    Polynomial generator(generator_coeff);          // generator polynomial g(x)
    Polynomial information;                         // Information bits I(x)
    Polynomial codeword;                            // Codeword C(x) (with error and erasure)
    Polynomial Answer;                              // correct codewrod A(x)
    Polynomial received_vector;                     // received vector R(x)
    Polynomial error;                               // Error E(x)
    Polynomial syndrome;                            // Syndrome S(x)
    Polynomial s0, s1, t0, t1;                      // Polynomial for EEA
    Polynomial tmp;                                 // temporary polynomial
    Polynomial I0, I1;                              // sigma0(x) and its formal derivative
    Polynomial W;                                   // error evaluator polynomial omega(x)
    Polynomial xr;                                  // x^r
```



```
// Encoding
    // Initilization
    information.degree = k - 1;
    Answer.degree = n - 1;
    codeword.degree = n - 1;
    received_vector.degree = n - 1;
    error.degree = n - 1;
    syndrome.degree = r - 1;
    I0.degree = 1;
    tmp.degree = 1;

    // Generate testcase
    outFile.open("testcase.txt");
    for(int testcase = 0; testcase < 100; testcase++){
        for(int i = 0; i <= k - 1; i++) information.data[i]  = rand() % (n + 1);
        codeword = information * generator;                  // C(x) = I(x) * g(x)
        for(int i = 0; i <= n - 1; i++) outFile << codeword.data[i] << " ";
        outFile << endl;
        for(int i = 0; i <= n - 1; i++) {                    // Add error and erasure
            int randi = rand() % 10;
            if(randi == 0) outFile << rand() % 64 << " ";
            else if(randi == 1) outFile << "*" << " ";
            else outFile << codeword.data[i] << " ";
        }
        outFile << endl;
    }
    outFile.close();

// Decoding
    inFile.open("testcase.txt");
    string s;
    for(int testcase = 0; testcase < 10; testcase++){
        // Initialization
        e0 = 0; e1 = 0;
        Answer.degree = n - 1;
        codeword.degree = n - 1;
        received_vector.degree = n - 1;
        error.degree = n - 1;
        syndrome.degree = r - 1;
        xr.degree = r;
        tmp.degree = 1;
        I0 = 1;
```

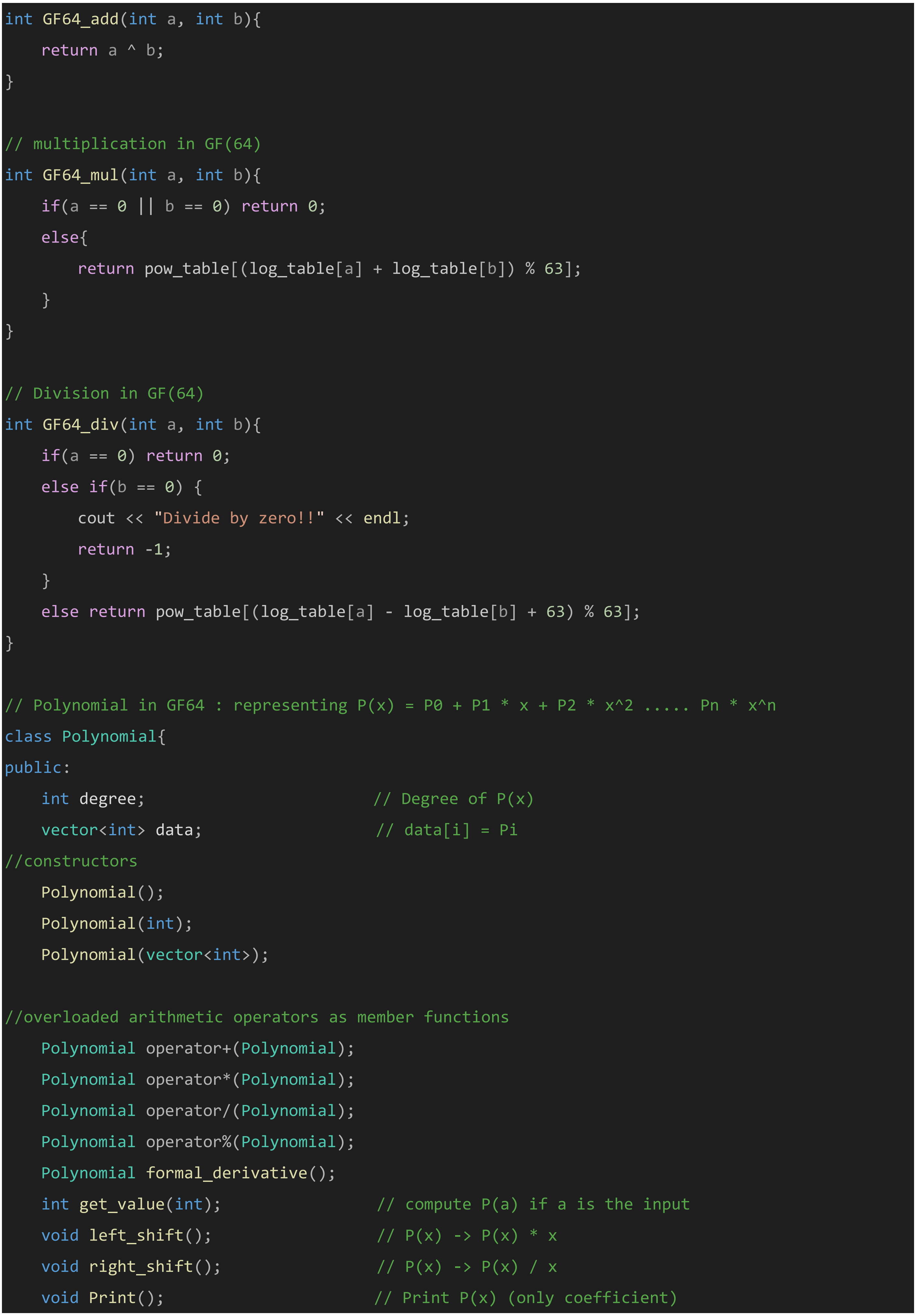


```cpp
        s0 = 1; s1 = 0;
        t0 = 0; t1 = 1;

        for(int i = 0; i <= n - 1; i++){
            error.data[i] = 0;
            syndrome.data[i] = 0;
            xr.data[i] = 0;
        }
        xr.data[r] = 1;
        for(int i = 0; i <= n - 1; i++) inFile >> Answer.data[i]; Answer.Print();
        for(int i = 0; i <= n - 1; i++){                       // Compute R'(x) and sigma0(x)
            inFile >> s;
            cout << s << " ";
            if(s == "*") {
                tmp.data[0] = 1; tmp.data[1] = pow_table[i];
                I0 = I0 * tmp;
                e0++;                                           // e0 = # of erasure
                codeword.data[i] = 0;
            }
            else {
                codeword.data[i] = stoi(s);
                if(codeword.data[i] != Answer.data[i]) e1++;
            }
        }
        cout << endl;
        cout << "e0 = " << e0 << "; e1 = " << e1 << endl;
        if(e0 > r) {
            cout << "failure" << endl;
            continue;
        }
        received_vector = codeword;
        // modulo g(x) for faster calculation of R(alpha^i)
        received_vector = received_vector % generator;

        // Compute Syndrome
        for(int i = 0; i <= r - 1; i++) {           // Sj = R(alpha^j)
            syndrome.data[i] = received_vector.get_value(pow_table[i + 1]);
        }
        syndrome.degree = r - 1;
        while(syndrome.data[syndrome.degree] == 0) syndrome.degree--;
        syndrome = (syndrome * I0) % xr;            // S0(x) = sigma0(x) * S(x) (mod x^r)
        EEA(xr, syndrome, s0, s1, t0, t1, W, e0); // Perform EEA to find sigma1(x) and omega(x)
```

```
                                                   // t1 = sigma1(x) and W = omega(x)
        I0 = I0 * t1;                              // sigma(x) = sigma0(x) * sigma1(x)
        I1 = I0.formal_derivative();               // Compute the formal derivative

        // Time Domain Approach
        failure = false;   // boolean variable for decode failure or not
        if(I0.data[0] == 0 || W.degree >= e0 + t1.degree) failure = true;
        else{
            count = 0;
            for(int i = 0; i <= n - 1; i++){
                x = GF64_div(1, pow_table[i]);
                if(I0.get_value(x) == 0 && I1.get_value(x) != 0){
                    count++;
                    // Ei = -omega(alpha ^ -i) / sigma'(alpha ^ -i)
                    error.data[i] = GF64_div(W.get_value(x), I1.get_value(x));
                }
                else error.data[i] = 0;
            }
            if(count != I0.degree) failure = true;
        }
        if(failure) cout << "failure!" << endl;
        else{
            codeword = codeword + error;                           // C(x) = R(x) - E(x)
            for(int i = 0; i <= n - 1; i++){                       // Compare C(x) with A(x)
                if(codeword.data[i] != Answer.data[i]){
                    codeword.Print();
                    Answer.Print();
                    system("pause");
                }
            }
            cout << "Testcase " << testcase << " pass!" << endl;
        }
        system("pause");
    }
    inFile.close();
    return 0;
}

void EEA(Polynomial P, Polynomial Q, Polynomial& s0, Polynomial& s1, Polynomial& t0,
Polynomial& t1, Polynomial& W, int e0){
    int u = (r - e0) / 2;                       // u = ceil(r - e0 / 2)
    int v = r - 1 - u;                          // v = r - 1 - u
    Polynomial q, tmp, tmps, tmpt;
    while(Q.degree > v || t1.degree > u){      // terminate condition : deg(rj(x) <= v) and 
deg(vj(x)) <= u
        q = P / Q;                             // q = P / Q
        tmp = P % Q;                           // r_j+1 = r_j-1 - q * r_j
        P = Q;
        Q = tmp;
        tmps = s0 + q * s1;                    // u_j+1 = u_j-1 - q * u_j
        s0 = s1; s1 = tmps;
        tmpt = t0 + q * t1;                    // v_j+1 = v_j-1 - q * v_j
        t0 = t1; t1 = tmpt;
    }
    W = Q;
}
```

b. Polynomial

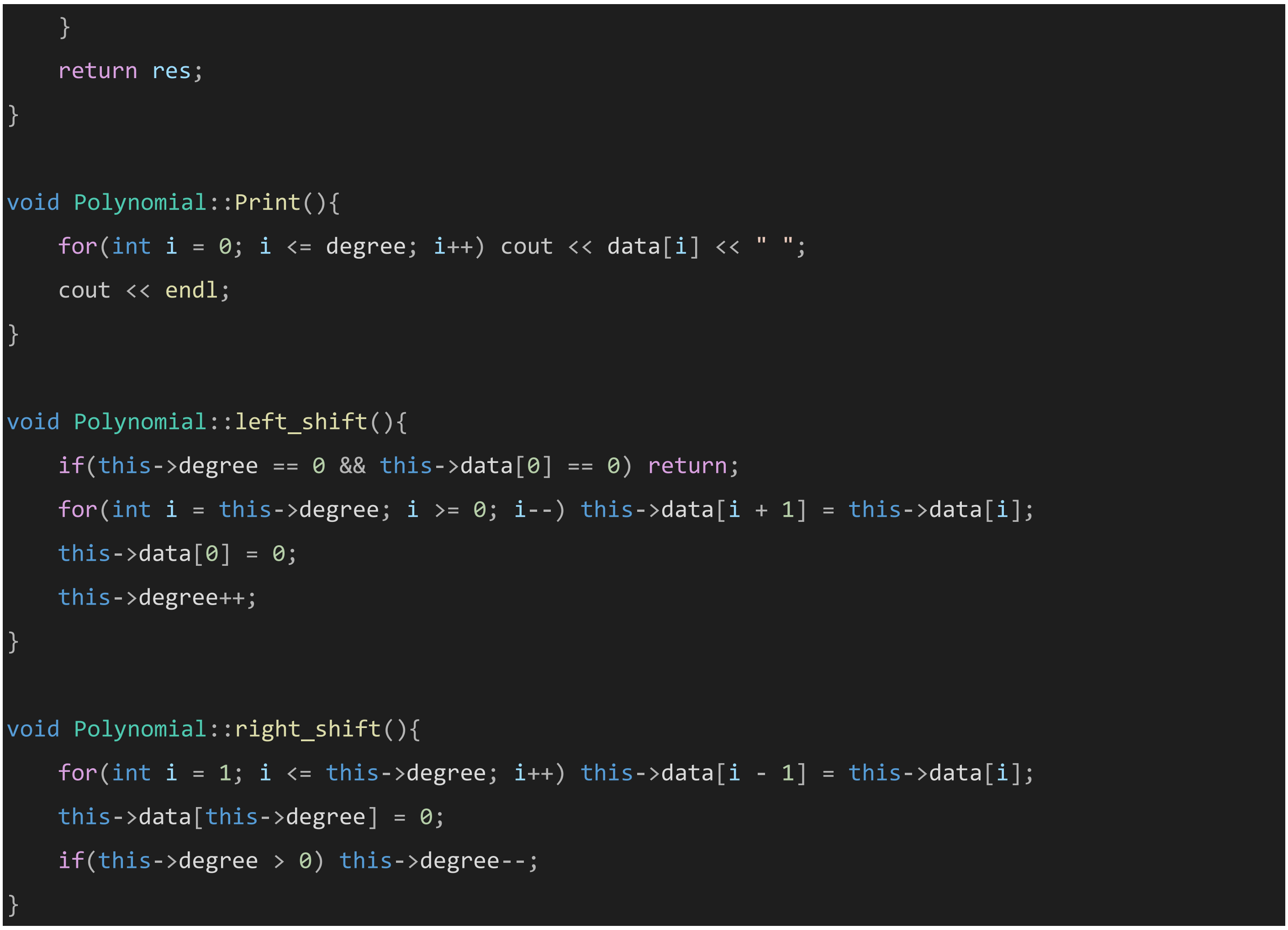
```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <string.h>

using namespace std;

#define MAX_Bit 1000
#define n 63
#define k 42
#define r 21

// GF(64) with a is a primitive element satisfying a^6 + a + 1

// pow_table[i] = a ^ i
vector<int> pow_table = {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};

// log_table[i] = log_a i with log_a 0 = -1
vector<int> log_table = {-1, 0, 1, 6, 2, 12, 7, 26,
                       3, 32, 13, 35, 8, 48, 27,
                       18, 4, 24, 33, 16, 14, 52,
                       36, 54, 9, 45, 49, 38, 28,
                       41, 19, 56, 5, 62, 25, 11,
                       34, 31, 17, 47, 15, 23, 53,
                       51, 37, 44, 55, 40, 10, 61,
                       46, 30, 50, 22, 39, 43, 29,
                       60, 42, 21, 20, 59, 57, 58};

// coefficient of generator polynomial
vector<int> generator_coeff = {58, 62, 59, 7, 35, 58, 63, 47, 51, 6, 33,
                              43, 44, 27, 7, 53, 39, 62, 52, 41, 44, 1};

// Addition in GF(64)
```



```cpp
int GF64_add(int a, int b){
    return a ^ b;
}

// multiplication in GF(64)
int GF64_mul(int a, int b){
    if(a == 0 || b == 0) return 0;
    else{
        return pow_table[(log_table[a] + log_table[b]) % 63];
    }
}

// Division in GF(64)
int GF64_div(int a, int b){
    if(a == 0) return 0;
    else if(b == 0) {
        cout << "Divide by zero!!" << endl;
        return -1;
    }
    else return pow_table[(log_table[a] - log_table[b] + 63) % 63];
}

// Polynomial in GF64 : representing P(x) = P0 + P1 * x + P2 * x^2 ..... Pn * x^n
class Polynomial{
public:
    int degree;                        // Degree of P(x)
    vector<int> data;                  // data[i] = Pi
//constructors
    Polynomial();
    Polynomial(int);
    Polynomial(vector<int>);

//overloaded arithmetic operators as member functions
    Polynomial operator+(Polynomial);
    Polynomial operator*(Polynomial);
    Polynomial operator/(Polynomial);
    Polynomial operator%(Polynomial);
    Polynomial formal_derivative();
    int get_value(int);                // compute P(a) if a is the input
    void left_shift();                 // P(x) -> P(x) * x
    void right_shift();                // P(x) -> P(x) / x
    void Print();                      // Print P(x) (only coefficient)
```

```
};

Polynomial::Polynomial(){
    degree = 0;
    data.assign(MAX_Bit, 0);
    for(int i = 0; i < MAX_Bit; i++) data[i] = 0;
}

Polynomial::Polynomial(int x){
    degree = 0;
    data.assign(MAX_Bit, 0);
    data[0] = x;
}

Polynomial::Polynomial(vector<int> d){
    degree = d.size() - 1;
    data.assign(MAX_Bit, 0);
    for(int i = 0; i <= degree; i++) data[i] = d[i];
}

Polynomial Polynomial::operator+(Polynomial y){     // res(x) = A(x) + B(x)
    Polynomial res;
    int degree;
    int x_len = this->degree;
    int y_len = y.degree;
    for(degree = 0; degree <= x_len || degree <= y_len; degree++){
        // res[i] = A[i] + B[i]
        res.data[degree] = GF64_add(this->data[degree], y.data[degree]);
    }
    while(degree >= 1 && res.data[degree] == 0) {  // check prefix zero and update degree
        degree--;
    }
    res.degree = degree;
    return res;
}

Polynomial Polynomial::operator*(Polynomial y){    // res(x) = A(x) * B(x)
    Polynomial res;
    int x_len = this->degree;
    int y_len = y.degree;
    int degree = x_len + y_len;
    for(int i = 0; i <= y_len; i++){                 // res[i] = sum(A[j] * B[i - j])
```

```
        for(int j = 0; j <= x_len; j++) {
            res.data[i + j] = GF64_add(res.data[i + j], GF64_mul(this->data[j], y.data[i]));
        }
    }
    while(degree >= 1 && res.data[degree] == 0) { // check prefix zero and update degree
        degree--;
    }
    res.degree = degree;
    return res;
}

Polynomial Polynomial::operator/(Polynomial y){  // A(x) = B(x) * res(x) + t(x)
    Polynomial t, tmp, res;
    if(this->degree < y.degree) return res;       // if deg(A(x)) < deg(B(x)) then res(x) = 0

    int i;
    int r_len = 0;
    t.degree = y.degree;
    for(i = 0; i <= y.degree; i++){               // 長除法 (long division)
        t.data[y.degree - i] = this->data[this->degree - i];
    }
    while(true){
        if(t.degree == y.degree){
            res.data[0] = GF64_div(t.data[t.degree], y.data[y.degree]);
            t = t + y * res.data[0];
        }
        if(i <= this->degree){
            t.left_shift();
            t.data[0] = this->data[this->degree - i];
            res.left_shift();
            i++;
        }
        else break;
    }
    return res;
}

Polynomial Polynomial::operator%(Polynomial y){ // A(x) = B(x) * q(x) + res(x)
    Polynomial t, tmp, res;
    if(this->degree < y.degree) return *this;    // if deg(A(x)) < deg(B(x)) then res(x) = A(x)

    int i;
    int q;
    int r_len = 0;
    t.degree = y.degree;
    for(i = 0; i <= y.degree; i++){                // 長除法 (long division)
        t.data[y.degree - i] = this->data[this->degree - i];
    }
    while(true){
        if(t.degree == y.degree){
            res.data[0] = GF64_div(t.data[t.degree], y.data[y.degree]);
            t = t + y * res.data[0];
        }
        if(i <= this->degree){
            t.left_shift();
            t.data[0] = this->data[this->degree - i];
            res.left_shift();
            i++;
        }
        else break;
    }
    return t;
}

Polynomial Polynomial::formal_derivative(){                  // A(x) -> A'(x)
    Polynomial res;
    if(this->degree == 0) return res;
    else{
        res.degree = this->degree - 1;
        for(int i = 0; i <= res.degree; i++){                // An * x^n -> n * An * x^(n - 1)
            // GF(64) has characteristic 2
            if(i % 2 == 0) res.data[i] = this->data[i + 1];
            else res.data[i] = 0;
        }
        while(res.data[res.degree] == 0) res.degree--;
        return res;
    }
}

int Polynomial::get_value(int alpha){                                // Compute A(alpha)
    int pow = alpha;
    int res = data[degree];                                          // Horner's rule
    for(int i = degree - 1; i >= 0; i--){
        res = GF64_add(data[i], GF64_mul(res, pow));
```



```
    }
    return res;
}

void Polynomial::Print(){
    for(int i = 0; i <= degree; i++) cout << data[i] << " ";
    cout << endl;
}

void Polynomial::left_shift(){
    if(this->degree == 0 && this->data[0] == 0) return;
    for(int i = this->degree; i >= 0; i--) this->data[i + 1] = this->data[i];
    this->data[0] = 0;
    this->degree++;
}

void Polynomial::right_shift(){
    for(int i = 1; i <= this->degree; i++) this->data[i - 1] = this->data[i];
    this->data[this->degree] = 0;
    if(this->degree > 0) this->degree--;
}
```