Polynomial Data Structures in C and C++

1 Introduction

This document summarizes how to build polynomial data structures in both C++ and C, and how to implement common operations such as:

- Representation (dense vs. sparse)
- Addition and subtraction
- Multiplication
- Evaluation at a point

In C++, we use `std::vector` for dense polynomials and `std::map` for sparse polynomials. In C, we build a linked-list-based representation.

2 Polynomials in C++ (Dense Representation)

A dense polynomial representation stores all coefficients from degree 0 up to some maximum degree in a contiguous array or `std::vector`. The index corresponds to the exponent:

$$\text{coefficients}[i] = a_i \quad \Rightarrow \quad a_i x^i.$$

2.1 Basic Dense Polynomial Class

```cpp
#include <vector>
#include <iostream>
#include <initializer_list>
#include <cmath>    // for std::pow

class Polynomial {
private:
    // coefficients[i] is the coefficient of x^i
    std::vector<double> coefficients;

public:
    // Constructor from a list of coefficients (lowest degree first)
    Polynomial(std::initializer_list<double> coeffs)
        : coefficients(coeffs) {}

    // Constructor for a specific degree (all coefficients initialized to 0)
    explicit Polynomial(int degree)
        : coefficients(degree + 1, 0.0) {}

    // Get the degree (highest non-zero exponent, or -1 for the zero polynomial)
    int degree() const {
        int d = static_cast<int>(coefficients.size()) - 1;
        while (d >= 0 && std::abs(coefficients[d]) == 0.0) {
```

```
            --d;
        }
        return d;
    }

    // Get coefficient of a specific power
    double getCoefficient(int power) const {
        if (power >= 0 && power < static_cast<int>(coefficients.size())) {
            return coefficients[power];
        }
        return 0.0; // Return 0 for non-existent terms
    }

    // Set coefficient of a specific power
    void setCoefficient(int power, double value) {
        if (power < 0) return;
        if (power >= static_cast<int>(coefficients.size())) {
            coefficients.resize(power + 1, 0.0);
        }
        coefficients[power] = value;
    }

    // Evaluate polynomial at x
    double evaluate(double x) const {
        double result = 0.0;
        double xPower = 1.0; // x^0
        for (std::size_t i = 0; i < coefficients.size(); ++i) {
            result += coefficients[i] * xPower;
            xPower *= x;
        }
        return result;
    }

    // Display the polynomial in human-readable form
    void display(std::ostream& os = std::cout) const {
        bool firstTerm = true;
        for (int i = static_cast<int>(coefficients.size()) - 1; i >= 0; --i) {
            double c = coefficients[i];
            if (std::abs(c) > 0.0) {
                // Handle the sign
                if (!firstTerm) {
                    if (c > 0) {
                        os << " + ";
                    } else {
                        os << " - ";
                        c = -c;
                    }
                } else if (c < 0) {
                    os << "-";
                    c = -c;
                }

                // Print coefficient (omit 1 for non-constant terms)
                if (i == 0 || std::abs(c - 1.0) > 0.0) {
                    os << c;
                }

                // Print variable part
```

```
                if (i > 1) {
                    os << "x^" << i;
                } else if (i == 1) {
                    os << "x";
                }

                firstTerm = false;
            }
        }

        if (firstTerm) { // All coefficients were zero
            os << "0";
        }
        os << '\n';
    }

    // Addition: this + other
    Polynomial operator+(const Polynomial& other) const {
        int maxDegree = std::max(degree(), other.degree());
        Polynomial result(maxDegree);

        for (int i = 0; i <= maxDegree; ++i) {
            double a = getCoefficient(i);
            double b = other.getCoefficient(i);
            result.setCoefficient(i, a + b);
        }
        return result;
    }

    // Subtraction: this - other
    Polynomial operator-(const Polynomial& other) const {
        int maxDegree = std::max(degree(), other.degree());
        Polynomial result(maxDegree);

        for (int i = 0; i <= maxDegree; ++i) {
            double a = getCoefficient(i);
            double b = other.getCoefficient(i);
            result.setCoefficient(i, a - b);
        }
        return result;
    }

    // Multiplication: this * other
    Polynomial operator*(const Polynomial& other) const {
        int degA = degree();
        int degB = other.degree();
        if (degA < 0 || degB < 0) {
            return Polynomial(0); // zero polynomial
        }

        Polynomial result(degA + degB);
        for (int i = 0; i <= degA; ++i) {
            for (int j = 0; j <= degB; ++j) {
                double current = result.getCoefficient(i + j);
                result.setCoefficient(i + j,
                                      current + getCoefficient(i) * other.getCoefficient(j));
            }
        }
```

```
        return result;
    }
};
```

Example Usage

```
int main() {
    Polynomial p{1.0, 2.0, 3.0};   // 1 + 2x + 3x^2
    Polynomial q{0.0, -1.0, 1.0};  // -x + x^2

    Polynomial sum = p + q;
    Polynomial prod = p * q;

    std::cout << "p(x)   = "; p.display();
    std::cout << "q(x)   = "; q.display();
    std::cout << "p+q    = "; sum.display();
    std::cout << "p*q    = "; prod.display();
    std::cout << "p(2.0) = " << p.evaluate(2.0) << "\n";

    return 0;
}
```

3 Polynomials in C++ (Sparse Representation)

When most coefficients are zero, a sparse representation is more memory-efficient. We can use an ordered map from exponent to coefficient:

```
#include <map>
#include <iostream>
#include <cmath>  // std::pow

class PolynomialSparse {
private:
    // Map: exponent -> coefficient
    std::map<int, double> terms;

public:
    // Set coefficient of a specific power
    void setCoefficient(int power, double value) {
        if (power < 0) return;
        if (std::abs(value) == 0.0) {
            terms.erase(power);  // Remove if coefficient is zero
        } else {
            terms[power] = value;
        }
    }

    // Get coefficient of a specific power
    double getCoefficient(int power) const {
        auto it = terms.find(power);
        if (it != terms.end()) {
            return it->second;
        }
        return 0.0;
    }

    // Evaluate polynomial at x
```

```
    double evaluate(double x) const {
        double result = 0.0;
        for (const auto& [exp, coeff] : terms) {
            result += coeff * std::pow(x, exp);
        }
        return result;
    }

    // Display the polynomial
    void display(std::ostream& os = std::cout) const {
        bool firstTerm = true;

        // Iterate in reverse order of exponents for standard display
        for (auto it = terms.rbegin(); it != terms.rend(); ++it) {
            int power = it->first;
            double coeff = it->second;
            double c = coeff;

            if (std::abs(c) > 0.0) {
                if (!firstTerm) {
                    if (c > 0) {
                        os << " + ";
                    } else {
                        os << " - ";
                        c = -c;
                    }
                } else if (c < 0) {
                    os << "-";
                    c = -c;
                }

                if (power == 0 || std::abs(c - 1.0) > 0.0) {
                    os << c;
                }

                if (power > 1) {
                    os << "x^" << power;
                } else if (power == 1) {
                    os << "x";
                }

                firstTerm = false;
            }
        }

        if (firstTerm) {
            os << "0";
        }
        os << '\n';
    }

    // Addition
    PolynomialSparse operator+(const PolynomialSparse& other) const {
        PolynomialSparse result = *this;
        for (const auto& [exp, coeff] : other.terms) {
            double sum = result.getCoefficient(exp) + coeff;
            result.setCoefficient(exp, sum);
        }
```

```
        return result;
    }

    // Subtraction
    PolynomialSparse operator-(const PolynomialSparse& other) const {
        PolynomialSparse result = *this;
        for (const auto& [exp, coeff] : other.terms) {
            double diff = result.getCoefficient(exp) - coeff;
            result.setCoefficient(exp, diff);
        }
        return result;
    }

    // Multiplication
    PolynomialSparse operator*(const PolynomialSparse& other) const {
        PolynomialSparse result;
        for (const auto& [exp1, coeff1] : terms) {
            for (const auto& [exp2, coeff2] : other.terms) {
                int newExp = exp1 + exp2;
                double newCoeff = result.getCoefficient(newExp)
                                  + coeff1 * coeff2;
                result.setCoefficient(newExp, newCoeff);
            }
        }
        return result;
    }
};
```

4 Polynomials in C (Linked List Representation)

In C, a common approach is to represent a polynomial as a singly linked list of terms, each term storing a coefficient and an exponent.

4.1 Data Structures

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Term {
    int coefficient;
    int exponent;
    struct Term* next;
} Term;

typedef struct Polynomial {
    Term* head;
} Polynomial;

// Utility: create an empty polynomial
Polynomial createPolynomial(void) {
    Polynomial p;
    p.head = NULL;
    return p;
}
```

4.2 Creating and Inserting Terms

We create nodes with `createTerm` and insert them in sorted order by exponent (descending), combining like terms when exponents match.

```
Term* createTerm(int coeff, int exp) {
    Term* newTerm = (Term*)malloc(sizeof(Term));
    if (newTerm == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newTerm->coefficient = coeff;
    newTerm->exponent = exp;
    newTerm->next = NULL;
    return newTerm;
}

// Insert a term in descending order of exponents and combine like terms
void insertTerm(Polynomial* poly, int coeff, int exp) {
    if (coeff == 0) {
        return;
    }

    Term* newTerm = createTerm(coeff, exp);

    // If list is empty or new exponent is greater than head's exponent
    if (poly->head == NULL || exp > poly->head->exponent) {
        newTerm->next = poly->head;
        poly->head = newTerm;
        return;
    }

    Term* current = poly->head;
    Term* prev = NULL;

    while (current != NULL && current->exponent > exp) {
        prev = current;
        current = current->next;
    }

    // If an existing term with the same exponent is found
    if (current != NULL && current->exponent == exp) {
        current->coefficient += coeff;
        free(newTerm);

        if (current->coefficient == 0) {
            // Remove the term from the list
            if (prev == NULL) {
                poly->head = current->next;
            } else {
                prev->next = current->next;
            }
            free(current);
        }
    } else {
        // Insert new term between prev and current
        newTerm->next = current;
        if (prev == NULL) {
            poly->head = newTerm;
```

```
        } else {
            prev->next = newTerm;
        }
    }
}
```

4.3 Displaying and Evaluating a Polynomial

```
void displayPolynomial(const Polynomial* poly) {
    const Term* current = poly->head;
    int firstTerm = 1;

    if (current == NULL) {
        printf("0\n");
        return;
    }

    while (current != NULL) {
        int c = current->coefficient;
        int e = current->exponent;

        if (!firstTerm) {
            if (c > 0) {
                printf(" + ");
            } else {
                printf(" - ");
                c = -c;
            }
        } else if (c < 0) {
            printf("-");
            c = -c;
        }

        if (e == 0 || c != 1) {
            printf("%d", c);
        }

        if (e > 1) {
            printf("x^%d", e);
        } else if (e == 1) {
            printf("x");
        }

        firstTerm = 0;
        current = current->next;
    }

    printf("\n");
}

int powerInt(int base, int exp) {
    int result = 1;
    while (exp-- > 0) {
        result *= base;
    }
    return result;
}

```

```
int evaluatePolynomial(const Polynomial* poly, int x_value) {
    int result = 0;
    const Term* current = poly->head;

    while (current != NULL) {
        result += current->coefficient
                  * powerInt(x_value, current->exponent);
        current = current->next;
    }
    return result;
}
```

4.4 Addition and Multiplication

```
Polynomial addPolynomials(const Polynomial* p1, const Polynomial* p2) {
    Polynomial result = createPolynomial();

    const Term* t1 = p1->head;
    const Term* t2 = p2->head;

    // Merge-like traversal (lists are sorted by exponent descending)
    while (t1 != NULL && t2 != NULL) {
        if (t1->exponent > t2->exponent) {
            insertTerm(&result, t1->coefficient, t1->exponent);
            t1 = t1->next;
        } else if (t1->exponent < t2->exponent) {
            insertTerm(&result, t2->coefficient, t2->exponent);
            t2 = t2->next;
        } else {
            int sumCoeff = t1->coefficient + t2->coefficient;
            insertTerm(&result, sumCoeff, t1->exponent);
            t1 = t1->next;
            t2 = t2->next;
        }
    }

    // Append remaining terms
    while (t1 != NULL) {
        insertTerm(&result, t1->coefficient, t1->exponent);
        t1 = t1->next;
    }
    while (t2 != NULL) {
        insertTerm(&result, t2->coefficient, t2->exponent);
        t2 = t2->next;
    }

    return result;
}

Polynomial multiplyPolynomials(const Polynomial* p1, const Polynomial* p2) {
    Polynomial result = createPolynomial();

    for (Term* t1 = p1->head; t1 != NULL; t1 = t1->next) {
        for (Term* t2 = p2->head; t2 != NULL; t2 = t2->next) {
            int newCoeff = t1->coefficient * t2->coefficient;
            int newExp   = t1->exponent + t2->exponent;
            insertTerm(&result, newCoeff, newExp);
        }
```

```
    }

    return result;
}
```

4.5 Memory Management

```
void freePolynomial(Polynomial* poly) {
    Term* current = poly->head;
    while (current != NULL) {
        Term* next = current->next;
        free(current);
        current = next;
    }
    poly->head = NULL;
}
```

Example Usage in C

```
int main(void) {
    Polynomial p = createPolynomial();
    Polynomial q = createPolynomial();

    // p(x) = 3x^2 + 2x + 1
    insertTerm(&p, 3, 2);
    insertTerm(&p, 2, 1);
    insertTerm(&p, 1, 0);

    // q(x) = x^3 - x + 4
    insertTerm(&q, 1, 3);
    insertTerm(&q, -1, 1);
    insertTerm(&q, 4, 0);

    printf("p(x) = ");
    displayPolynomial(&p);

    printf("q(x) = ");
    displayPolynomial(&q);

    Polynomial sum = addPolynomials(&p, &q);
    printf("p(x) + q(x) = ");
    displayPolynomial(&sum);

    Polynomial prod = multiplyPolynomials(&p, &q);
    printf("p(x) * q(x) = ");
    displayPolynomial(&prod);

    int x = 2;
    printf("p(%d) = %d\n", x, evaluatePolynomial(&p, x));

    freePolynomial(&p);
    freePolynomial(&q);
    freePolynomial(&sum);
    freePolynomial(&prod);

    return 0;
}
```

5 Summary

- In C++, dense polynomials can be represented using `std::vector<double>` and sparse polynomials using `std::map<int,double>`.
- Operations such as addition, subtraction, multiplication, and evaluation can be implemented by iterating over coefficients or terms and combining exponents appropriately.
- In C, a linked-list-based representation provides a flexible way to store polynomial terms with dynamic insertion and combination of like terms.

These patterns give you a solid foundation for polynomial manipulation in both C and C++.