# 解題說明:

這個 C++ 程式實現了一個多項式類別 Polynomial，使用循環鏈表來表示多項式。以下是程式的主要結構和功能：

• Node 類別：

• 用於表示多項式中的每個項，包含係數 (coef)、指數 (exp)、以及指向下一個節點的連結 (link)。

• Polynomial 類別：

• 使用循環鏈表表示多項式，每個節點代表一項多項式。

• 實現了輸入輸出運算子、複製建構子、賦值運算子、解構子，以及多項式的加法、減法、乘法、評估等功能。

# Algorithm Design & Programming：

#include <iostream>

#include <cmath>

#include <limits>

class Node {

public:

int coef; // 係數

int exp; // 指數

Node\* link; // 下一個節點的指標

};

class Polynomial {

private:

Node\* header; // 多項式的鏈表頭節點

static Node\* av; // 可用節點的鏈表頭節點

public:

// 建構子

Polynomial();

// 解構子

~Polynomial();

// 輸入運算子

friend std::istream& operator>>(std::istream& is, Polynomial& x);

// 輸出運算子

friend std::ostream& operator<<(std::ostream& os, const Polynomial& x);

// 複製建構子

Polynomial(const Polynomial& a);

// 賦值運算子

const Polynomial& operator=(const Polynomial& a);

// 多項式相加

Polynomial operator+(const Polynomial& b) const;

// 多項式相減

Polynomial operator-(const Polynomial& b) const;

// 多項式相乘

Polynomial operator\*(const Polynomial& b) const;

// 多項式評估

float Evaluate(float x) const;

};

// 初始化可用節點的鏈表頭節點

Node\* Polynomial::av = nullptr;

// 多項式的建構子

Polynomial::Polynomial() {

header = new Node;

header->link = header;

}

// 輸入多項式的運算子

std::istream& operator>>(std::istream& is, Polynomial& x) {

int n;

is >> n; // 多項式項數

Node\* current = x.header;

for (int i = 0; i < n; ++i) {

Node\* newNode = new Node;

is >> newNode->coef >> newNode->exp;

newNode->link = nullptr;

current->link = newNode;

current = newNode;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current->link = x.header;

return is;

}

// 輸出多項式的運算子

std::ostream& operator<<(std::ostream& os, const Polynomial& x) {

Node\* current = x.header->link; // 跳過頭節點

int count = 0;

while (current != x.header) {

os << (count > 0 ? " + " : "") << current->coef;

if (current->exp > 0) {

os << "x";

if (current->exp > 1)

os << "^" << current->exp;

}

current = current->link;

++count;

}

return os;

}

// 多項式複製建構子的實作

Polynomial::Polynomial(const Polynomial& a) {

Node\* current\_a = a.header->link; // 跳過 a 的頭節點

header = new Node; // 初始化新的頭節點

Node\* current\_this = header;

while (current\_a != a.header) {

Node\* newNode = new Node;

newNode->coef = current\_a->coef;

newNode->exp = current\_a->exp;

newNode->link = nullptr;

current\_this->link = newNode;

current\_this = newNode;

current\_a = current\_a->link;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current\_this->link = header;

}

// 賦值運算子的實作

const Polynomial& Polynomial::operator=(const Polynomial& a) {

if (this != &a) {

// 清除原有節點

Node\* current = header->link; // 跳過頭節點

while (current != header) {

Node\* temp = current;

current = current->link;

delete temp;

}

// 複製新的多項式

Node\* current\_a = a.header->link; // 跳過 a 的頭節點

Node\* current\_this = header;

while (current\_a != a.header) {

Node\* newNode = new Node;

newNode->coef = current\_a->coef;

newNode->exp = current\_a->exp;

newNode->link = nullptr;

current\_this->link = newNode;

current\_this = newNode;

current\_a = current\_a->link;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current\_this->link = header;

}

return \*this;

}

// 解構子的實作

Polynomial::~Polynomial() {

Node\* current = header->link; // 跳過頭節點

while (current != header) {

Node\* temp = current;

current = current->link;

delete temp;

}

// 釋放頭節點

delete header;

}

// 多項式相加的實作

Polynomial Polynomial::operator+(const Polynomial& b) const {

Polynomial result;

Node\* current\_a = header->link; // 跳過頭節點

Node\* current\_b = b.header->link; // 跳過b的頭節點

Node\* current\_result = result.header;

while (current\_a != header || current\_b != b.header) {

Node\* newNode = new Node;

if (current\_a->exp == current\_b->exp) {

newNode->coef = current\_a->coef + current\_b->coef;

newNode->exp = current\_a->exp;

current\_a = current\_a->link;

current\_b = current\_b->link;

}

else if (current\_a->exp > current\_b->exp) {

newNode->coef = current\_a->coef;

newNode->exp = current\_a->exp;

current\_a = current\_a->link;

}

else {

newNode->coef = current\_b->coef;

newNode->exp = current\_b->exp;

current\_b = current\_b->link;

}

newNode->link = nullptr;

current\_result->link = newNode;

current\_result = newNode;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current\_result->link = result.header;

return result;

}

// 多項式相減的實作

Polynomial Polynomial::operator-(const Polynomial& b) const {

Polynomial result;

Node\* current\_a = header->link; // 跳過頭節點

Node\* current\_b = b.header->link; // 跳過b的頭節點

Node\* current\_result = result.header;

while (current\_a != header || current\_b != b.header) {

Node\* newNode = new Node;

if (current\_a->exp == current\_b->exp) {

newNode->coef = current\_a->coef - current\_b->coef;

newNode->exp = current\_a->exp;

current\_a = current\_a->link;

current\_b = current\_b->link;

}

else if (current\_a->exp > current\_b->exp) {

newNode->coef = current\_a->coef;

newNode->exp = current\_a->exp;

current\_a = current\_a->link;

}

else {

newNode->coef = -current\_b->coef;

newNode->exp = current\_b->exp;

current\_b = current\_b->link;

}

newNode->link = nullptr;

current\_result->link = newNode;

current\_result = newNode;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current\_result->link = result.header;

return result;

}

// 多項式相乘的實作

Polynomial Polynomial::operator\*(const Polynomial& b) const {

Polynomial result;

Node\* current\_a = header->link; // 跳過頭節點

Node\* current\_result = result.header;

while (current\_a != header) {

Node\* current\_b = b.header->link; // 跳過b的頭節點

while (current\_b != b.header) {

Node\* newNode = new Node;

newNode->coef = current\_a->coef \* current\_b->coef;

newNode->exp = current\_a->exp + current\_b->exp;

newNode->link = nullptr;

// 將新節點插入結果多項式

Node\* temp = current\_result->link;

Node\* prev = current\_result;

while (temp != result.header && temp->exp > newNode->exp) {

prev = temp;

temp = temp->link;

}

if (temp != result.header && temp->exp == newNode->exp) {

// 同次幂的項相加

temp->coef += newNode->coef;

delete newNode;

}

else {

// 插入新節點

prev->link = newNode;

newNode->link = temp;

current\_result = newNode; // 修正此行

}

current\_b = current\_b->link;

}

current\_a = current\_a->link;

}

// 將最後一個節點的 link 指向頭節點，形成循環鏈表

current\_result->link = result.header;

return result;

}

// 多項式評估的實作

float Polynomial::Evaluate(float x) const {

float result = 0.0;

Node\* current = header->link; // 跳過頭節點

while (current != header) {

result += current->coef \* std::pow(x, current->exp);

current = current->link;

}

return result;

}

int main() {

Polynomial p1, p2, p3;

std::cout << "輸入第一個多項式:\n";

std::cin >> p1;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "輸入第二個多項式:\n";

std::cin >> p2;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

p3 = p1 \* p2;

std::cout << "兩多項式相乘的結果: " << p3 << std::endl;

return 0;

}

# 效能分析：

空間複雜度：

多項式的表示使用循環鏈表，因此節點的數量取決於多項式的項數。空間複雜度主要由項數和節點的數量決定。

時間複雜度：

加法、減法、乘法等運算的時間複雜度受到多項式的項數和每項的指數的影響。

類別中的複製建構子和賦值運算子的實作使用了循環鏈表的複製，時間複雜度也與項數和指數相關。

## 測試與驗證：

# 心得感想:

寫完這個 C++ 程式後，我覺得自己對多項式運算有了更深的理解。透過循環鏈表來表示多項式，讓我在處理加減乘法時，能更靈活地操作每個項。雖然寫起來有點複雜，但看到程式能夠正確運行並計算出多項式的結果，還是很有成就感。不過也發現，隨著多項式的項數增加，時間和空間的需求也會變高，這是未來可以再優化的地方。